Engenharia de Segurança - Trabalho 2

Álvaro José Lopes
10873365

Natan Henrique Sanches
11795680

Gabriel da Cunha Dertoni
11795717

Osni Brito de Jesus
11857330

Pedro Lucas de Moliner de Castro
11795784

Introdução

Este trabalho foi realizado pelo grupo da disciplina de Engenharia de Segurança com o propósito de aprender mais sobre como funcionam atackes cibernéticos na prática. O caso de estudo escolhido para este trabalho foi o de *binary exploitation*, em particular em *buffer overflow*. Quanto ao programa vítima, foi escolhido um progama de complexidade considerável. O programa é destinado à utilizar o Universal Chess Interface (UCI) [1] para confrontar diferentes *engines* de xadrez.

Encontrando a vulnerabilidade

Com acesso ao código fonte do programa, foi possível verificar que haviam diversas faltas de verificações, principalmente relacionadas à verificação de condições de borda. Em particular uma função chama atenção:

```
bool uci_read_command(UciCommand* out) {
    char command[UCI_MAX_CMD_SIZE + 1];
    ASSERT(scanf("%s", command) != EOF);
    if (!uci_parse_command_kind(command, &out->kind)) return false;

    /*...*/
}
```

Essa função lê da entrada padrão um comando do protocolo UCI e escreve o resultado em `out`. Caso encontre um erro, ela retorna `false`. O buffer `command` é grande o suficiente para armazenar qualquer comand UCI válido, entretanto não há garantias de que a entrada será somente tão grande quando o maior comando UCI. Portanto, é possível dar *overflow* nesse buffer, o que pode fazer a aplicação parar de funcionar corretamente.

Tomando proveito da vulnerabilidade

Por conta da presença da vulnerabilidade descrita na seção anterior, podemos nos aproveitar do fato desse buffer existir somente na `stack` para sobreescrever o endereço de retorno da função. Dessa maneira, em tese é possível inserir um endereço qualquer no lugar do endereço de retorno original e controlar o fluxo de código de acordo com a vontade do atacante.

A técnica utilizada para manipular os endereços de retorno é chamada *Return Oriented Programming* (ROP). Primeiro, para saber como sobreescrever o endereço de retorno, é necessário descobrir quantos bytes se encontram entre a variável `char command[]` e o endereço de retorno. Para isso, basta acesso a uma cópia do binário e acesso a um debugger. Neste caso essa distância é de 56 bytes. Então preenchemos esses primeiros 56 bytes com lixo e depois podemos inserir um endereço em código para onde redirecionar o fluxo.

Idealmente gostaríamos de conseguir acesso à uma shell na máquina atacada. Para isso, podemos utilizar a função `execv("/bin/sh", 0)` da libc. Mas para conseguir chamar essa função, antes é necessário descobrir o endereço dela. Como essa não é uma função utilizada pelo programa original, não podemos utilizar da *Procedure Linkage Table*. Ao invés disso é necessário descobrir o endereço onde foi carregada a libc e daí descobrir o endereço da função `execv` a partir de seu endereço relativo dentro da libc.

Isso se faz necessário pois SOs modernos utilizam de *Address Space Layout Randomization* (ASLR), que propositalmente carrega bibliotecas dinâmicas em endereços distintos a cada execução para prevenir esse tipo de ataque. Entretanto, basta descobrir o endereço absoluto de uma única função da libc para descobrir o endereço onde ela foi carregada. Por sorte, podemos usar da função `puts` que já possui seu endereço na *Global Offset Table* (GOT) e uma referência na PLT, uma vez que é usada dentro do programa original. Então, para extrair o endereço da função `puts` podemos utilizar ela própria para imprimir o próprio endereço. Para fazer isso, basta passar o endereço da GOT como argumento para a `puts`. E por sua vez, para isso precisamos colocar o endereço da `puts` na `GOT` no registrador `rdi`, que em arquiteturas x86_64, é o primeiro argumento de acordo com a conveção de chamada. Para então tirar um valor da stack e colocá-lo em `rdi` podemos usar de *gadgets*. Os *gadgets* são pedaços de código tipicamente com alguma manipulação de registrador imediatamente seguido de uma instrução `ret`. No caso, o *gadget* que precisamos é `pop %rdi; ret`. Por sorte a ferramenta `pwntools` [2] já nos fornece com funções prontas para encontrar qualquer *gadget* necessário. Por fim, após vazar o endereço, podemos pular novamente para `uci_read_command` e executar mais ROPs de lá.

Uma vez montado, o primeiro ROP (com comentários):

```
0x0000:         0x403fb3 ; pop rdi; ret
0x0008:         0x406018 ; [arg0] rdi = got.puts
0x0010:         0x401f70 ; puts_flush
0x0018:         0x403fb3 ; pop rdi; ret
0x0020:              0x0
0x0028:         0x403c77 ; uci_read_command
```

Após executar o ROP, o programa deve imprimir o endereço da função `puts`. Com esse endereço, é trivial descobrir o endereço da libc, basta subtrair o endereço relativo da função `puts` do endereço absoluto encontrado. Com esse endereço podemos executar o segundo ROP, o que de fato chama `execv("/bin/sh", 0)`. A string `"/bin/sh"` já se encontra na libc, então podemos simplesmente usar seu endereço e carragá-lo em `rdi` usando um gadget. Também é necessário carregar `0` em `rsi`, o segundo argumento. Por fim, chamamos a função `execv` de dentro da libc. O segundo ROP (com comentários):

```
0x0000:   0x7f9c06baa01f ; pop rsi; ret
0x0008:              0x0 ; [arg1] rsi = 0
0x0010:   0x7f9c06ba7b6a ; pop rdi; ret
0x0018:   0x7f9c06d385bd ; [arg0] rdi = "/bin/sh"
0x0020:   0x7f9c06c672d0 ; execv
```

Por fim, ao executar esse ROP, o programa executa uma shell, e o atacante ganha acesso à máquina que está executando a aplicações.

Como prevenir esse ataque

Proteções em nível de SO, como ASLR, dificultam mas não impossibilitam esse tipo de ataque, como foi visto. Além disso compiladores tipicamente adicionam um protetor de stack próximo às instruções de retorno das funções, o que previne esse tipo de ROP. Esse segurança está ativada por padrão na versão do gcc utilizada e, portanto, foi desativada através da flag `-fno-stack-protector` por motivos educacionais. Além disso, *Position Independent Executables* também são mais difíceis de abusar dessa maneira, visto que mesmo as funções do prórpio programa teriam seu endereço randomizado. Essa proteção também foi desabilitada para testes através da flag `-no-pie`.

Por fim, o mais importante é evitar ao máximo o uso de buffers sem verficação dos casos de borda. Já existem algumas flags que ajudam em casos similares, como de uso de `fgets`. Entretanto, mesmo com as flags `-Wall -Werror -pedantic`, o `gcc` não foi capaz de detectar a vulnerabilidade citada. Portanto, é de suma importância que programadores tenham ciência dos riscos de utilizar buffers limitados na stack sem verificar seu limite de tamanho.

Etapas para reproduzir

1. Acesse o código em https://github.com/GabrielDertoni/binary-exploitation
2. Siga as instruções do README.md

Notas finais

É possível que mudar de compilador ou de versão do compilador (utilizado foi `gcc-11.1.0`) altere os resultados. Além disso, mesmo utilizando o mesmo compilador também houveram ocasiões onde o exploit não funcionava. Então vale a pena tentar algumas iterações de compilação.

Bibliografia

[1] "UCI protocol." https://backscattering.de/chess/uci/

[2] "Pwntools." https://github.com/Gallopsled/pwntools