Exploit / Stack Buffer Overflow

Phish'n'Chips Team

Agenda

- General background
  - What is exploitation (in CTF context)?
  - Real-world examples
  - Exploitation techniques and defenses
- Stack buffer overflow
  - Basic mechanism (overwriting value)
  - Overwriting function pointer
  - Overwriting return address

Disclaimer - ethical use of the content

- This material is solely for the purpose of security education, especially for ethical security researchers, security contest participants and penetration testers. The readers agree that they **do not use the technique for malicious purposes**, such as intruding into other people's computers without the owners' content.
- Disclosing exploitation techniques with a malicious intent is a criminal offense in some countries. The author declares that **the material is created without the aim of misuse or encouraging misuse**, and consists of only publicly available techniques. In case any damage occurs by misuse of this document, the person who used the technique against the law takes complete responsiblility for the consequences, and such act or damage shall not be considered grounds for denying legality of this document under any law in any country.

What is exploitation (in CTF context)?

- Taking advantage of (typically binary) program
- Execute arbitrary code

Real-world examples

CVE-2019-0708 (BlueKeep)

> A remote code execution vulnerability exists in Remote Desktop Services formerly known as Terminal Services when an unauthenticated attacker connects to the target system using RDP and sends specially crafted requests, aka 'Remote Desktop Services Remote Code Execution Vulnerability'.
> NVD - CVE-2019-0708 https://nvd.nist.gov/vuln/detail/CVE-2019-0708

Exploit: https://github.com/rapid7/metasploit-framework/blob/master/modules/exploits/windows/rdp/cve_2019_0708_bluekeep_rce.rb

Real-world examples

For who don't have knowledge in cybersecurity / software exploitation:

> Oops, super-hackers can use *magic* to get into my computer and steal my name, postal address, money and whatever they like! 😱

```
# part of exploit code by @zerosum0x0
def eternalblue_kshellcode_x64(process="spoolsv.exe"):
    return 'U\xe8b\x00\x00\x00\xb9\x82\x00\x00\xc0\x0f2L\x8d\rh\x00\x00\x00D9\xc8t\x199E\x00t\n\x89U\x04\x89E
    \x00\xc6E\xf8\x00I\x91PZH\xc1\xea \x0f0]eH\x8b\x04%\x88\x01\x00\x00f\x83\x80\xc4\x01\x00\x00\x01L\x8d\x9c
    ...
    \xff\xd0H\x83\xc40\xc3'
```

This is of course not true - actually there are many techniques which can be used for exploitation (and defense). Each technique has strengths and limitations, and should be combined cleverly to achieve successful exploitation.

There is no problem if we don't understand them now - let's learn one by one 🙂

Exploitation techniques and defenses

Basic major techniques

- Attack surface (What vulnerability can attackers take advantage of?)
  - Buffer overflow (stack buffer overflow CWE-121, heap overflow CWE-122)
  - Format string vulnerability CWE-134
- Code execution (How attackers can achieve their intent?)
  - Important variable overwrite
  - Function pointer overwrite (pointer variable, vtbl, GOT)
  - Return address overwrite
  - Shellcode execution
  - Return-to-libc, Return-oriented programming (ROP)

Exploitation techniques and defenses

Defense techniques (not exaustive)

- mostly applied nowadays
  - NX bit (W⊕X memory model) (vs. shellcode execution)
  - Stack canary (vs. stack buffer overflow)
  - ASLR (Address Space Layout Randomization) (vs. function pointer / return address overwrite)
  - checked memory allocator (vs. heap overflow)
- available now, but not widely used
  - FORTIFY_SOURCE (vs. format string attack)
  - control flow integrity (vs. ROP)

Exploitation techniques and defenses

The topic we focus on this slide deck:

- Attack surface (What vulnerability can attackers take advantage of?)
  - Buffer overflow (**stack buffer overflow** CWE-121, heap overflow CWE-122)
  - Format string vulnerability CWE-134
- Code execution (How attackers can achieve their intent?)
  - **Important variable overwrite**
  - **Function pointer overwrite** (**pointer variable**, vtbl, GOT)
  - **Return address overwrite**
  - Shellcode execution
  - Return-to-libc, Return-oriented programming (ROP)

stack buffer overflow - basic mechanism

What is a stack?

```
+--------------+ 0x00400000
| code region |
+--------------+ 0x00401fff

+--------------+ 0x00600000
| data region |
+--------------+ 0x00610000
| heap         |
+--------------+ 0x0061ffff

+--------------+ 0x7fff0000
| stack        |
+--------------+ 0x7fffffff
```

Program executes using memory: code region, data region, heap, and stack.

- code region: machine instructions
- data region: global variables
- heap: dynamically allocated data (malloc)
- stack: temporary data in function scope (local variables)

stack buffer overflow - basic mechanism

Compiler normally allocates stack region for local variables sequentially

```c
void f()
{
    int x = 1;
    int y = 2;
    int z = 3;
    char a[4] = "AAAA";
    char b[5] = "BBBB";
    ...
}
```

```
Stack
+--------------+          +--------------+
| x = 1        |          | b = "BBBB"   |
+--------------+          +--------------+
| y = 2        |          | a = "AAAA"   |
+--------------+    or    +--------------+
| z = 3        |          | z = 3        |
+--------------+          +--------------+
| a = "AAAA"   |          | y = 2        |
+--------------+          +--------------+
| b = "BBBB"   |          | x = 1        |
+--------------+          +--------------+
```

stack buffer overflow - basic mechanism

Reading into buffer in incorrect size may result in buffer overflow

```c
void f()
{
    char x  [8] = "AAAAAAAA";
    char buf[8] = "BUFFER  ";
    char z  [8] = "CCCCCCCC";
    read(0, buf, 16); // should be 8
    write(1, x, 8); write(1, "\n", 1);
    if (memcmp(x, "AAAAAAAA", 8) != 0) {
        puts("You win!");
    }
}
```

```
Stack
    +-----------------+
z   | C C C C C C C C |
    +-----------------+ read()
buf | B U F F E R     |   |
    +-----------------+   v
x   | A A A A A A A A |
    +-----------------+
```

stack buffer overflow - 📝 hands-on (1)

Try to overflow the buffer to print `You win!`

```c
void f()
{
    char x   [8] = "AAAAAAAA";
    char buf[8] = "BUFFER  ";
    char z   [8] = "CCCCCCCC";
    read(0, buf, 16); // should be 8
    write(1, x, 8); write(1, "\n", 1);
    if (memcmp(x, "AAAAAAAA", 8) != 0) {
        puts("You win!");
    }
}
```

```
Stack
    +-----------------+
z   | C C C C C C C C |
    +-----------------+ read()
buf | B U F F E R     |   |
    +-----------------+   v
x   | A A A A A A A A |
    +-----------------+
```

stack buffer overflow - numbers, strings and endianness

We need to understand data format / layout to understand exploitation

How data is stored in the memory?

- Strings
  - stored sequentially, followed by "NUL" character (byte `\x00`)
  - each byte represented as an ASCII code

```c
char buf[16] = "Hello World!";
```

```
+----+----+----+----+----+----+----+----+
| 48 | 65 | 6c | 6c | 6f | 20 | 57 | 6f | H e l l o _ W o
+----+----+----+----+----+----+----+----+
| 72 | 6c | 64 | 21 | 00 | 00 | 00 | 00 | r l d ! . . . .
+----+----+----+----+----+----+----+----+
           (_ represents space, and . represents NUL here)
```

stack buffer overflow - numbers, strings and endianness

How data is stored in the memory?

- Numbers
  - normally stored as a fixed length of bytes (4 bytes, 8 bytes, ...)
  - normally integers have hexadecimal, 2's complement representation
  - normally real numbers have floating point representation
  - In x86, the last byte (8bit = 2 hexadecimall chars) of hexadecimal representation are stored first (**little endian**)

```
int x = 1234567890; // 32-bit 0x499602D2

/*    +----+----+----+----+
    x | d2 | 02 | 96 | 49 |
      +----+----+----+----+ */

long long y = 1122334455667788LL; // 64-bit 0x3FCC1DA8C9C4C

/*    +----+----+----+----+----+----+----+----+
    y | 4c | 9c | 8c | da | c1 | fc | 03 | 00 |
      +----+----+----+----+----+----+----+----+ */
```

stack buffer overflow - numbers, strings and endianness

Data handling in programs

- C programs
  - type safety - compile error when interpreted as a wrong type
  - we can force "unsafe" operations by type cast
- binary programs
  - no types - binary data can be interpreted either as integers, floating points, strings

```
int  x[2] = { 0x414243, 0x646566 };
char s[8] = "Hello";

/*  +-------------------------+
  x | 43 42 41 00 66 65 64 00 |
    +-------------------------+
  s | 48 65 6c 6c 6f 00 00 00 |
    +-------------------------+ */

puts(s);          // -> "Hello"
puts(x);          // compile error
puts((char*)x);   // -> "CBA"
```

stack buffer overflow - 📝 hands-on (2)

What will be the result of the following program?

Run the program to see the result.

```c
#include <stdio.h>

int main(void) {
  int  x[2] = { 0x414243, 0x646566 };
  char s[8] = "Hello";

  printf("%s %d\n", (char*)x, x[0]);
  printf("%s %d\n", s, *(int*)s);
  return 0;
}
```

PHISH N CHIPS
CTF TEAM

stack buffer overflow - 📝 hands-on (3)

Try to overflow the buffer to print `You win!`

```c
void f()
{
    int  x  [2] = { 0x1122, 0x3344 };
    char buf[8] = "BUFFER";
    int  z  [2] = { 0x5566, 0x7788 };

    read(0, buf, 16); // should be 8

    if (x[0] == 1231234123) {
        puts("You win!");
    }
}
```

```
Stack
+-----------------------------+
| 66 55 00 00   88 77 00 00 |
+-----------------------------+ read()
| B  U  F  F    E  R  00 00 |    |
+-----------------------------+    v
| 22 11 00 00   44 33 00 00 |
+-----------------------------+
```

Hint1: `1231234123 == 0x????????`

Hint2: Be careful of byte order

stack buffer overflow - function pointer

- function pointer
  - C function mechanism to change program behavior dynamically
  - In most platform, a function pointer is the address of the beginning of the function to be called
  - binary representation: same as 8 bytes integer in 64bit x86, and 4 bytes integer in 32bit x86

```c
typedef void (*logger_func_t)(const char*);

void file_log(const char *s) {
    static FILE *logfile = fopen("logfile.log", "r");
    fprintf(logfile, "%s\n", s);
}
void stdout_log(const char *s) {
    puts(s);
}
void f(logger_func_t log) {
    log("DEBUG: f is called");
    // ...
}
int main() {
    logger_func_t log = &stdout_log;
    if (option.log_to_file) {
        log = &file_log;
    }
    // ...
    f(log);
    // ...
}
```

stack buffer overflow - function pointer

How can we dentify the function address?

```c
int f() {
}

int main() {
    printf("address of f = %p\n", &f);
    return 0;
}


// $ ./program
// address of f = 0x4004e7
```

- objdump: disassembler

```
$ objdump -d ./program
...
00000000004004e7 <f>:
  4004e7:       55                      push   %rbp
  4004e8:       48 89 e5                mov    %rsp,%rbp
...
```

- readelf: obtain Linux program binary (ELF format) metadata

```
$ readelf -s ./program
...
  45: 0000000000601030   0 NOTYPE  GLOBAL DEFAULT  23 _edata
  46: 0000000000400594   0 FUNC    GLOBAL DEFAULT  14 _fini
  47: 00000000004004e7   7 FUNC    GLOBAL DEFAULT  13 f
...
```

stack buffer overflow - function pointer

If the read data overflows into function pointer, the subsequent call to function pointer will be modified

```c
void dummy() {
    puts("Nope");
}
void win() {
    puts("Win!");
}
struct S {
  char buf_a[8];  // 8 byte buffer
  void (*fun)();  // 8 byte function pointer
  char buf_b[8];  // 8 byte buffer
};
void f() {
  struct S x = { "BUFFER_A", &dummy, "BUFFER_B" };
  scanf("%s", x.buf_a);
  printf("%s\n", x.buf_a);
  x.fun();
}
```

```
Stack
+-------------------------+ scanf()
| B  U  F  F  E  R  _  A |    |
+-------------------------+    v
| func ptr: dummy         |      <- called
+-------------------------+
| B  U  F  F  E  R  _  B |
+-------------------------+
```

```
Stack
+-------------------------+
| A  A  A  A  A  A  A  A |
+-------------------------+
| crafted address         |      <- called
+-------------------------+
| B  U  F  F  E  R  _  B |
+-------------------------+
```

stack buffer overflow - 📝 hands-on (4)

Try to overwrite the function pointer, and print `Win!`

```c
void dummy() {
    puts("Nope");
}
void win() {
    puts("Win!");
}
struct S {
  char buf_a[8];  // 8 byte buffer
  void (*fun)();  // 8 byte function pointer
  char buf_b[8];  // 8 byte buffer
};
void f() {
  struct S x = { "BUFFER_A", &dummy, "BUFFER_B" };
  scanf("%s", x.buf_a);
  printf("%s\n", x.buf_a);
  x.fun();
}
```

```
Stack
+-------------------------+ scanf()
| B  U  F  F  E  R  _  A |    |
+-------------------------+    v
| func ptr: dummy         |      <- called
+-------------------------+
| B  U  F  F  E  R  _  B |
+-------------------------+
```

stack buffer overflow - return address

- C function calls
  - functions can be called from arbitrary location in the program
  - how the function can identify and return to its caller?
- return address
  - program stores a code address to which the program should return (jump)
  - In x86, the return address is stored on stack

```
int f(int x) {
    return x * x; // look at stored return address
                  // and jump to that address
                  // (L1 or L2 or ...)
}

void g() {
    x = f(123); // store return address L1
                // and jump to f
L1: printf("%d\n", x);
}

void h() {
    y = f(456); // store return address L2
                // and jump to f
L2: printf("0x%x\n", y);
}
```

stack buffer overflow - return address

- stack layout (x86)
  - return address is stored between local variables and parameters (in 64bit x86, parameters are normally stored in registers)
  - *stack pointer* (`rsp` or `rbp`) points to the top of local variables
  - conventionally, the *base pointer* (`rbp` or `ebp`) stores a pointer between local variables and return address

```
+----------------+ <- stack pointer
| local var 1    |
+----------------+
| local var 2    |
+----------------+
| ...            |
+----------------+ <- base pointer
| (stored bp)    |
+----------------+
| return address |
+----------------+
| func param 1   |
+----------------+
| func param 2   |
+----------------+
| ...            |
+----------------+
```

stack buffer overflow - return address

Overflow buffer to overwrite return address

```c
void win() {
    puts("Win!");
    execl("/bin/sh", "/bin/sh", NULL);
}

void func() {
    char x[8] = "ABCDEFGH";
    char y[8] = "XXXXXXXX";
    char z[8] = "IJKLMNOP";

    scanf("%s", y);
    return;
}
```

```
Stack
  +-------------------------+
z | I  J  K  L  M  N  O  P |
  +-------------------------+ scanf()
y | X  X  X  X  X  X  X  X |   |
  +-------------------------+   |
x | A  B  C  D  E  F  G  H |   |
  +-------------------------+   v
  | ...                     |
  +-------------------------+
  | return address          | <-- jump to
  +-------------------------+   this addr
  | ...                     |
  +-------------------------+
```

stack buffer overflow - 📝 hands-on (5)

Try to overwrite return address, and execute `win` function!

```c
void win() {
    puts("Win!");
    execl("/bin/sh", "/bin/sh", NULL);
}
void func() {
    char x[8] = "ABCDEFGH";
    char y[8] = "XXXXXXXX";
    char z[8] = "IJKLMNOP";

    printf("What's your name? > ");
    fflush(stdout);
    scanf("%s", y);
    printf("Hello, %s!\n", y);
    return;
}
```

```
Stack
   +-------------------------+
z  | I  J  K  L  M  N  O  P  |
   +-------------------------+ scanf()
y  | X  X  X  X  X  X  X  X  |    |
   +-------------------------+    |
x  | A  B  C  D  E  F  G  H  |    |
   +-------------------------+    v
   | ...                     |
   +-------------------------+
   | return address          | <-- jump to
   +-------------------------+   this addr
   | ...                     |
   +-------------------------+
```

stack buffer overflow - writing scripts to exploit

There are several ways to feed binary data to the program:

- creating binary data and using pipe
  - ```
    $ echo -ne '\x9e\xf3\x02\x00' | ./victim
    ```
  - cannot handle input/output interactively
- writing script
  - write custom program (with C, python, ...) to interact with program
  - use **pwntools** - recommended way for writing exploits for CTF challenges
    ```

stack buffer overflow - writing scripts to exploit

- Prerequisites
  - install python3 and pwntools

```
$ pip3 install pwntools
```

```
from pwn import *
context.arch = 'amd64'        # set architecture (i386, amd64)

p = process('./challenge')  # run program
print(p.recvline())           # read line from program
p.send(b'\x9e\xf3\x02\x00') # write binary data
p.send(p32(0x12345678))       # send binary data encoded with correct endian
p.interactive()               # hand over to interactive session
```

See also: http://docs.pwntools.com/en/stable/

stack buffer overflow - writing scripts to exploit

- Typical CTF challenge in exploitation category
  - a vulnerable program is running on a remote machine, listening on a TCP port
  - "flag" file is placed on the remote machine
  - exploit the vulnerability to run `/bin/sh`, then read the flag
  - e.g. if challenge says `pwn.example.com 1337`, connect to TCP port 1337 of `pwn.example.com`
- Use pwntools `remote` feature instead of `process` (interface is almost the same)

```
if remote:
    p = remote('pwn.example.com', 1337)
else:
    p = process('./challenge')
```

stack buffer overflow - 📝 hands-on (6)

Solve manually-solved challenges with pwntools

- 📝 hands-on (3) - overflow into integer variable
- 📝 hands-on (4) - overflow into function pointer variable
- 📝 hands-on (5) - overflow into return address

stack buffer overflow - history and references

- Morris Worm (1988)
  - one of the first major malware samples spread across the Internet
  - exploits stack buffer overflow vulnerability of finger daemon
  - Mark W. Eichin, Jon A. Rochis. With Microscope and Tweezers: An Analysis of the Internet Virus of November 1988, In Proc of IEEE S&P, 1989

    > The virus hit the finger daemon (`fingerd`) by overflowing a buffer which was allocated on the stack. ... the overflow allowed a fake stack frame to be created, which caused a small piece of code to be executed when the procedure returned.

PHISH N CHIPS CTF TEAM

stack buffer overflow - history and references

- Aleph One (Elias Levy). Smashing The Stack For Fun And Profit. Phrack magazine, issue 49(14), 1996
- Jonathan Pincus, Brandon Baker. Beyond stack smashing: recent advances in exploiting buffer overruns. In [J] IEEE Secur. Priv. 2(4), 2004

stack buffer overflow - securing software

- Secure SDLC (Software Development Life Cycle)
- Development and testing
  - Objective: reduce the number of vulnerabilities
  - Secure coding
  - Vulnerability testing, for example:
    - Source code analysis
    - Binary code analysis
    - Fuzz testing (Fuzzing)
- Runtime mitigation (explained below)
  - Objective: make potential vulnerabilities hard to exploit

stack buffer overflow - runtime mitigation

- Stack canary
  - put a randomly-generated value (stack canary) in front of return address
  - verify the value just before returning
  - if the buffer overflow rewrites return address, it also rewrites stack canary, and the verification will fail
  - thus, attacker needs to guess stack canary value, which is generally impossible

```
+----------------+ write +----------------+
| buffer         |   |   | overwrittenXXX |
+----------------+   |   +----------------+
| ...            |   |   | overwrittenXXX |
+----------------+   |   +----------------+
| stack canary   |   |   | overwrittenXXX |
+----------------+   |   +----------------+
| return address |   v   | crafted-addr   |
+----------------+       +----------------+
```

stack buffer overflow - runtime mitigation

- ASLR (Address Space Layout Randomization)
  - countermeasure to function pointer overwrite / return pointer overwrite
  - randomize code and data address, so that attackers cannot guess the correct jump target address
  - Scope of address randomization
    - stack address (activated in most cases)
    - shared library address (activated in most cases)
    - main program address (sometimes activated as of 2020)
  - PIE (position independent executable) - ASLR applied to main program address

stack buffer overflow - runtime mitigation

Inspect which countermeasures are used by pwntools

```
$ pwn checksec ./program
[*] '/path/to/program'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
```

- Stack: `Canary found` or `No canary found`
- PIE: `PIE Enabled` or `No PIE (0x<baseaddr>)`

stack buffer overflow - takeaway

- What's learned
  - memory / stack layout (return address, local variables)
  - data representation (strings, integers)
  - overwriting data by buffer overflow
  - overwriting function pointer and return address by buffer overflow
  - using `pwntools` to write an exploit
  - countermeasures to stack buffer overflow (stack canary, PIE)
- What comes next
  - Shellcode execution
  - Return-to-libc / ROP
  - Dealing with countermeasures

Thank you for listening!

Questions? 🙂

PHISH N CHIPS
CTF TEAM