Advanced Security for Systems Engineering – VO 04: Advanced Attacks on Applications 2

Clemens Hlauschek, Daniel Marth, Christian Schanes
Capture-the-Flag Team defragmented.brains

- Take part in many international hacking competitions
- Diverse bunch, different skills and skill levels
- Join our mailinglist: ctf-join@inso.tuwien.ac.at
- Next CTF: HITCON 25.-27.11.
Memory Corruption Bugs: Basics
Memory Corruption Bugs: Results of successful exploits

- **Denial of Service**
  - Induce process crash, prevent clients from accessing service

- **Information Disclosure**
  - Leaking private information (e.g., passwords, private keys)
  - Often 1st step in circumventing mitigation techniques (e.g., leaking process space address information)

- **Control Flow Hijacking**
  - Maliciously alter the process' behaviour: “Arbitrary Code Execution”
Memory Corruption Bugs: Control Flow Hijacking

1. Modify control flow data / metadata with user input
   - Function return address
   - Function pointer
   - Virtual method table
   - Heap metadata
   - Global Offset Table (GOT) or Import Address Table (IAT)

2. Redirect Control Flow
   - to injected (machine) code
   - or to existing code in the process’ memory space
Process Execution

Registers (x86):
- %eip: points to next instruction being executed
- %esp: points to end of stack

Stack: store for information about currently active subroutine
Function gray pushes parameters for function yellow on the stack.
Stack Layout: During Function Call

```
void gray ()
{
    ...
    yellow(a1,a2);
    ...
}

int yellow(int p1, int p2)
{
    char buf[3];
    int l1, l2;
    l2 = blue(l1, buf);
    return l2;
}
```

- Call instruction pushes %eip register (return address) on the stack
Stack Layout: Function Prologue

```c
void gray()
{
   ...
   yellow(a1, a2);
   ...
}

int yellow(int p1, int p2)
{
   char buf[3];
   int l1, l2;
   l2 = blue(l1, buf);
   return l2;
}
```

- Save gray’s frame pointer (%ebp)
- Update frame pointer
- Save callee-saved registers
## Stack Layout: Function Prologue

```c
void gray() {
    ...
    yellow(a1, a2);
    ...
}

int yellow(int p1, int p2) {
    char buf[3];
    int l1, l2;
    l2 = blue(l1, buf);
    return l2;
}
```

- Allocate local variables
Stack Layout: Function Prologue

```c
void gray()
{
    ...
    yellow(a1, a2);
    ...
}

int yellow(int p1, int p2)
{
    char buf[3];
    int l1, l2;
    l2 = blue(l1, buf);
    return l2;
}
```

- Local Variables allocated traditionally in order of declaration
Stack Layout: Before Function Call

```c
void gray()
{
    ...
    yellow(a1, a2);
    ...
}

int yellow(int p1, int p2)
{
    char buf[3];
    int l1, l2;
    l2 = blue(l1, buf);
    return l2;
}
```

- Save caller-saved registers
- Push params for blue
void gray()
{
    ... 
    yellow(a1, a2);
    ...
}

int yellow(int p1, int p2)
{
    char buf[3];
    int l1, l2;

    l2 = blue(l1, buf);
    return l2;
}
Stack Layout: After Function Call

```c
void gray()
{
    ...
    yellow(a1, a2);
    ...
}

int yellow(int p1, int p2)
{
    char buf[3];
    int l1, l2;
    l2 = blue(l1, buf);
    return l2;
}
```

- Stack frame for blue and params have already been freed (by incrementing %esp)
void gray ()
{
    ...  
    yellow (a1, a2);  
    ... 
}

int yellow (int p1, int p2)
{
    char buf [3];  
    int l1, l2; 
    l2 = blue (l1, buf);  
    return l2; 
}
Stack Layout: After Function Call

```c
void gray()
{
    ...
    yellow(a1, a2);
    ...
}

int yellow(int p1, int p2)
{
    char buf[3];
    int l1, l2;
    l2 = blue(l1, buf);
    return l2;
}
```

... and execution continues right after the call to `yellow` in function `gray`
**Stack Layout: Purpose of Frame Pointer**

```c
void gray() {
    ...
    yellow(a1, a2);
    ...
}

int yellow(int p1, int p2) {
    char buf[3];
    int l1, l2;
    l2 = blue(l1, buf);
    return l2;
}
```

- Positive offset added to `%ebp` to address parameter
- Negative offset added to `%ebp` to address local variable
Stack Layout: Frame Pointer

```c
void gray()
{
    ...
    yellow(a1, a2);
    ...
}

int yellow(int p1, int p2)
{
    char buf[3];
    int l1, l2;

    l2 = blue(l1, buf);
    return l2;
}
```

- `%ebp` can often be optimized away
- `gcc: -fomit-frame-pointer`
Stack Buffer Overflow: Vulnerable Program

```c
int main(int argc, char** argv)
{
    char val;
    val = yellow(5, argv[1]);
    printf("auth: %c", val);
}

int yellow(int len, char* text)
{
    char authenticated = 'N';
    char buf[3];
    ...  
    strcpy(buf, text);
    ...  
    return authenticated;
}
```

val = yellow(...
printf("aut ...
return 0;

text: argv[1]
len: 5
return address
main's ebp
callee-saved regs
authenticated: 'N'

char buf[3]; strcpy(...
return auth...

caller-saved regs
src: text
dest: &buf[0]
return address
yellow's ebp
....
Stack Buffer Overflow: Local Variable Spill

- buf[3] overflows with user input "123Y"
- "Y" spills into variable authenticated
Stack Buffer Overflow: Disrupt Control Flow

What if we spill input further up the stack?

- Return address gets overwritten
- Program segfaults after function `yellow` tries to return

```
$ gcc -fno-stack-protector -g -o vuln vuln.c
$ ./vuln 123YAAAAAAAAAAAAAAAA
Segmentation fault
$ 
```
Stack Buffer Overflow: Stack Content Before Overflow

Stack content right before call to `strcpy`

- At 0xbffff36c the char array `buf` starts
- At 0xbffff37c the original return address is stored
- At 0xbffff378 the frame pointer of `main` is stored

```bash
gdb$ run AAAABBBBBCCCCDDDEEEEEE
Breakpoint 1, 0x080484dd in yellow (len=0x5, text=0xbffff5e9 "AAAAABBBBBCCCCDDDEEEEEE") at vuln.c:17
17      strcpy(buf, text);
gdb$ x/12x buf
0xbffff36c: 0x4e048354 0xb7ff1080 0x08049ff4 0xbffff3a8
0xbffff37c: 0x080484a5 0x00000005 0xbffff5e9 0x08048520
0xbffff38c: 0xbffff3a8 0xb7e91235 0xb7ff1080 0x0804852b
```
Stack content right after call to `strcpy`

- `buf` is filled with the string "AAA" (ascii code for 'A': 0x41)
- The rest of the input string "AAAABBBBCCCCDDDDDEEEE" overflows
- The original return address at 0xbffff37c is overwritten with 0x45454545
We can redirect control flow to (almost) arbitrary locations in the process’ memory space.
Inject own malicious code ("shellcode") into the process’ memory space:

- Provide the shellcode as part of the input string, it gets copied in the buffer buf
- Overwrite return address to point to the beginning of buf [3]

Achieved: arbitrary attacker-controlled computations.
Stack Buffer Overflow Attack: Summary

- Fill buffer with own code (shellcode)
- Overwrite return address
- Return address points to shellcode
- When leaving the current function
  - Overwritten return address will be loaded into %eip register (instruction pointer)
  - %eip register points to shellcode
  - Shellcode will be executed
Basic stack layout, a horizontal perspective

string grows

buffer

return address

stack grows
Stack Buffer Overflow: Recapitulation

String spills out of buffer, overwrites saved return address.
Stack Buffer Overflow: NOP-Sled

New return address needs to point to buffer: Exact location not known.

- Prepend **NOP-Sled** to shellcode as “landing zone”
- Make an educated guess for an address somewhere in the NOP-Sled
Literature / Links

- Richarte (2002): Four different tricks to bypass Stackshield and Stackguard protection.
- Corelan Team: Exploit writing tutorial part 6: Bypassing Stack Cookies, SafeSeh, SEHOP, HW DEP and ASLR.
Literature / Links

Thank you!

https://security.inso.tuwien.ac.at/