(Early) Memory Corruption Attacks

CS-576 Systems Security
Instructor: Georgios Portokalidis
Fall 2018
“Memory corruption occurs in a computer program when the contents of a memory location are unintentionally modified due to programming errors; this is termed violating memory safety.

When the corrupted memory contents are used later in that program, it leads either to program crash or to strange and bizarre program behavior. “

--wikipedia
Programs Are Deterministic

The program implements the functionality intended by the developer.
Programs Are Deterministic

Program functionality severely altered

Original program $\rightarrow$ arbitrary program based on input

Input B $\rightarrow$ Program $\rightarrow$ Output B

Input C $\rightarrow$ Program $\rightarrow$ Output C

Input D $\rightarrow$ Program $\rightarrow$ Output D

Unexpected or untested input triggering vulnerability in program

Malicious or garbage input
Incorrect handling of untested or incorrect input is one the main causes of software vulnerabilities.
Remote Vs. local

Local attacks
- If the user input can be only provided by a local user

Remote attack
- If the user input can be only provided over the network
Local Attacks

All programs run with the privileges of the running user (Effective UID)
Local Attacks

Input produced by another user

Program

Arbitrary program executes with the rights of the user executing it
Privileged Resources

User

Program

User accessible resources

Administrator

Program

Privileged resources
SETUID Programs

Programs that run with the privileges of their owner, not the executing user
Privilege Escalation Attacks

The arbitrary program executes with elevated privileges

Input produced by another user

Privileged resources

SETUID Program
Remote Attacks

An arbitrary program is run at a remote host

www.stevens.edu

Host: www
OS: Debian
HTTP Server: nginx
Remote Attacks

An arbitrary program is run at a remote host

Bad Input

www.stevens.edu

Host: www
OS: Debian
HTTP Server: nginxd

Running as root

With high privileges
Common Vulnerabilities

Overflows: Writing beyond the end of a buffer

Underflows: Writing beyond the beginning of a buffer

Use-after-free: Using memory after it has been freed

Uninitialized memory: Using pointer before initialization

Null pointer dereferences: Using NULL pointers

Type confusion: Assume a variable/object has the wrong type

HW errors: Hammering memory to cause bit flips to non-owned memory
CWE™ is a community-developed list of common software security weaknesses. It serves as a common language, a measuring stick for software security tools, and as a baseline for weakness identification, mitigation, and prevention efforts.

View the CWE List

View by Research Concepts  View by Development Concepts  View by Architectural Concepts

Search CWE

Easily find a specific software weakness by performing a search of the CWE List by keywords(s) or by CWE-ID Number. To search by multiple keywords, separate each by a space.

Google Custom Search

See the full CWE List page for enhanced information, downloads, and more.

Total Software Weaknesses: 7,14
Buffer Overflows
Buffer Overflows

Writing outside the boundaries of a buffer

Common programmer errors that lead to it ...

- Insufficient input checks/wrong assumptions about input
- Unchecked buffer size
- Integer overflows
Stack Overflows
int mytest(char *str)
{
    char buf[16];
    strcpy(buf, str);
    printf("%s\n", buf);
    return strlen(buf);
}
Stack Overflow Example

```c
int mytest(char *str)
{
    char buf[16];
    strcpy(buf, str);
    printf("%s\n", buf);
    return strlen(buf);
}
```
`int mytest(char *str) {
    char buf[16];
    strcpy(buf, str);
    printf("%s\n", buf);
    return strlen(buf);
}

./mytest AAAAA

High address/stack bottom

RETADDR
buf
buf
buf
buf

ACK

Low address/stack top`
int mytest(char *str)
{
    char buf[16];
    strcpy(buf, str);
    printf("%s\n", buf);
    return strlen(buf);
}

./mytest AAAAA
int mytest(char *str)
{
    char buf[16];
    strcpy(buf, str);
    printf("%s\n", buf);
    return strlen(buf);
}

./mytest AAAAAAAAAAAAAAAAAAAAAAAA
int mytest(char *str)
{
    char buf[16];
    strcpy(buf, str);
    printf("%s\n", buf);
    return strlen(buf);
}

./mytest AAAAAAAAAAAAAAAAAAAAAAAA
Control-Flow Hijacking

The saved return address is a code pointer stored in memory

- Controlling it grants control of a control-flow instruction (e.g., ret)

Untrusted inputs that lead to corruption of a code pointer lead to control-flow hijacking attacks
Other Code Pointers

| Return address | return; | ret |
| Function address | ```typedef void (*cmpf_t)(int, int);
void compare(int array[], int len, int num, cmpf_t f)
{
    int i;
    for (i < 0; i < len; i++)
        if (array[i] < num)
            f(i, array[i]);
} ``` | call *(rax)* |
| Jump table | switch (option) {
  case 0:
    Code ...
  case 1:
    Code ...
  ...
} | jmp *(rax)* |
Where to Point Execution

SHELLCODE
malicious machine code

Malicious injected code is also code shellcode, because the first instances where used to spawn a shell.
Injecting Shellcode

\(0??\)

SHELLCODE

buf + 0x14

AAAA

AAAA

AAAA

AAAA

AAAA
Code Injection

Code injection (CI) - Injecting machine code into a vulnerable program’s memory

Code injections attacks inject code and use control-flow hijacking to execute that code
Shellcode Limitations

Injected shellcode cannot include a null byte because of `strcpy()`

Shellcode needs to be carefully crafted to avoid disallowed bytes

Other methods of copying data may not have the same limitation: `memcpy()`, `gets()`, `read()`, `fread()`, custom copy routines, etc.
```c
static void getURL(void) {
    char buf[64];
    read(STDIN_FILENO, buf, 128);
    get_webpage(buf);
}
```

No limitation on bytes read.
int mytest(char *str)
{
    char buf[16];
    strcpy(buf, str);
    printf("%s\n", buf);
    return strlen(buf);
}
int mytest(char *str)
{
    char buf[16];
    strcpy(buf, str);
    printf("%s\n", buf);
    return strlen(buf);
}

./mytest AAAAAAAAAAAAAAAAAAAAA
int mytest(char *str)
{
    char buf[16];
    strcpy(buf, str);
    printf("%s\n", buf);
    return strlen(buf);
}

./mytest AAAAAAAAAAAAAAAA
int mytest(char *str)
{
    char buf[16];
    strcpy(buf, str);
    printf("%s\n", buf);
    return strlen(buf);
}

./mytest 'AAAAAAAAAAAAAAAAAAAAAA'

Function exit (LEAVE)

movl %ebp, %esp
pop %ebp
int mytest(char *str)
{
    char buf[16];
    strcpy(buf, str);
    printf("%s\n", buf);
    return strlen(buf);
}

./mytest AAAAAAAAAAAAAAAAA\x3c\xca\xff\xffAAAA

movl %ebp, %esp
pop %ebp

Function exit (LEAVE)