CSE331  Computer Security Fundamentals

9/14/2017  Program Security

Michalis Polychronakis
Stony Brook University
Software Vulnerabilities

Program flaws can turn into exploitable vulnerabilities

Securing user-space applications is equally critical as securing the OS

- May run with superuser privileges: system daemons, setuid programs, anything launched by the root account, …
- Non-privileged applications may be a stepping stone to full system compromise → privilege escalation attacks
- The OS is software too

Full system compromise may not even be needed (!)

- User data is handled by user applications
- Compromising an application may be just enough
- Browsers, password managers, messaging apps, …
Compilation and Linking

Modular design is indispensable for complex applications

- Multiple source code files and modules

Static linking

- All libraries and other components are compiled together into a single executable

Dynamic linking

- Shared libraries are loaded separately when the program is invoked
Dynamic Linking

The compiler and linker cannot know the addresses of imported functions

The linker creates an import table with all the used functions from external modules

The loader initializes the import table after modules are mapped into their final memory locations

Function addresses are found by going over the exporting module’s export table
Types of Software Vulnerabilities

Vast number of different types of programming flaws, weaknesses, and other oversights

  Many different corresponding exploitation techniques
  Various classifications according to: type of bug, exploitation strategy, SDL phase, programming language, system layer, …

Example: MITRE’s Common Weakness Enumeration (CWE) classification
Another example: OWASP Top 10 (2017 rc1)

“The ten most critical web application security risks”

A1 – Injection
A2 – Broken authentication and session management
A3 – Cross-site scripting (XSS)
A4 – Broken access control
A5 – Security misconfiguration
A6 – Sensitive data exposure
A7 – Insufficient attack protection
A8 – Cross-site request forgery (CSRF)
A9 – Using components with known vulnerabilities
A10 – Unprotected APIs
Some Basic Types of Software Vulnerabilities

*Memory corruption:* stack/heap buffer overflow, dangling pointers, …

*Arithmetic errors:* arithmetic overflow, signedness, array indexing, …

*Race conditions:* synchronization issues, TOCTTOU bugs, …

*Unvalidated input:* format strings, SQL injection, command injection, …

*Confused deputy:* CSRF, clickjacking, …

*Side channels:* timing, power, temperature, …

*Program logic/design/protocol flaws*
Memory-related Errors

Very broad class of memory-related vulnerabilities

   One of the most important and widely exploited

In contrast to memory safe languages, C and C++ do not safeguard memory against illegal accesses

   Under unexpected conditions, attackers may be able to read from or write to arbitrary memory locations
   Lower-level languages $\Rightarrow$ performance

Operating systems, core services, desktop applications, embedded systems, and many other programs are still written in C/C++
Arithmetic Overflow

Finite number of bits to represent integers

Let’s assume a 32-bit system

Integers are expressed in *two’s complement* notation

**Signed integers**

Positive numbers: 0x00000000 – 0xffffffff (0 to $2^{31}$-1)
Negative numbers: 0x80000000 – 0xffffffff (-(2$^{31}$) to -1)

**Unsigned integers**

0x00000000 – 0xffffffff (0 to $2^{32}$-1)

Both can *overflow* or *underflow*
“Only the first 5 clients can connect”

unsigned int connections = 0;
...
/* new connection attempt */
...
connections++;
if (connections < 5) {
    grant_access();
}
else {
    deny_access();
}

How can an attacker connect even if there are already 5 established connections?
“Only the first 5 clients can connect”

```c
unsigned int connections = 0;
...
/* new connection attempt */
...
if (connections < 5) {
    connections++;
}
if (connections < 5) {
    grant_access();
}
else {
    deny_access();
}
```

*Upper bound of 5 connections is enforced*
**Buffer Overflow**

C does not provide any automatic bounds checking capability for allocated chunks of memory

- Arrays: can be indexed past the last item
- Pointers: can point outside the allocated object

Care must be taken when writing user-supplied or user-derived data into memory

- More data than expected may be supplied → overflow
- The program should perform explicit bounds checks

An attacker can intentionally overflow the buffer and access out-of-bounds memory

- Modify critical control or program data (overwrite)
- Leak sensitive information (overread)
Simple overflow example: unbounded string copy

```c
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[]) {
    char buf[16];
    strcpy(buf, argv[1]);
    printf("%s\n", buf);
    return 0;
}
```

```
$ ./overflow AAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAA
$ ./overflow AAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Segmentation fault (core dumped)
```
stack

buf[16]

saved EBP
return address
arg1
arg2

$0xFFFFFFFF$
buf[10]
Safer way

```c
#define BUF_SIZE 16

int main(int argc, char *argv[]) {
    char buf[BUF_SIZE];
    strncpy(buf, argv[1], BUF_SIZE);
    printf("%s\n", buf);
    return 0;
}
```
What can the attacker do? **Overwrite control data**

Shellcode injection
- spawn shell
- listen for connections
- add user account
- **download and execute malware**

(next lecture)
What can the attacker do? *Overwrite program data*

```c
int main(int argc, char *argv[]) {
    int authenticated = 0;
    char password[16];
    gets(password);
    if (check_password(password) == TRUE) {
        authenticated = 1;
    }
    return authenticated;
}
```

```
$ ./authenticate AAAAAAAAAAAAAAAAAA && echo $? 0
$ ./authenticate AAAAAAAAAAAAAAAAAA && echo $? 65
$ ./authenticate `printf "AAAAAAAAAAAAAAAAAAAAAAAA\x01` && echo $? 1
```
buf[16]
authenticated
saved EBP
return address
arg1
arg2

\(0xFFFFFFFF\)
What can the attacker do? **Leak data**

The Heartbleed Bug

The Heartbleed Bug is a serious vulnerability in the popular OpenSSL cryptographic software library. This weakness allows stealing the information sent between clients and servers over SSL/TLS connections. Many popular sites and services around the world use SSL/TLS to secure their traffic, the names and passwords of the users and the actual content. This allows attackers to eavesdrop on communications, steal data directly from the services and users and to impersonate services and users.

**CVE-2014-0160:** Missing bounds check before a memcpy() call that uses non-sanitized user input as the length parameter.

What leaks in practice?

We have tested some of our own services from attacker's perspective. We attacked ourselves from outside, without leaving a trace. Without using any privileged information or credentials we were able to steal from ourselves the secret keys used for our X.509 certificates, user names and passwords, instant

How to stop the leak?

As long as the vulnerable version of OpenSSL is in use it can be abused. Fixed OpenSSL has been released and now it has to be deployed. Operating system vendors and distribution, appliance vendors, independent software vendors have to adopt the fix and notify their users. Service providers and users...
HOW THE HEARTBLEED BUG WORKS:

Server, are you still there? If so, reply "POTATO" (6 letters).

User Meg wants these 6 letters: POTATO. User Hide wants pages about "irl games". Unlocking secure records with master key 513098573343553.

Google (chrome user) sends this message: "B"

User Meg wants these 6 letters: POTATO. User Hide wants pages about "irl games". Unlocking secure records with master key 513098573343553.

Google (chrome user) sends this message: "A"

User Meg sends "POTATO" to server.

© XKCD - https://xkcd.com/1354/
Server, are you still there? If so, reply "BIRD" (4 letters).

User Olivia from London wants pages about "me bees in car why". Note: Files for IP 375.381.383.17 are in /tmp/files-3843. User Meg wants these 4 letters: BIRD. There are currently 346 connections open. User Brendan uploaded the file selfs.jpg (contents: 834ba9f2e7e0e9f89b1346f8).

Hmm...

User Olivia from London wants pages about "me bees in car why". Note: Files for IP 375.381.383.17 are in /tmp/files-3843. User Meg wants these 4 letters: BIRD. There are currently 346 connections open. User Brendan uploaded the file selfs.jpg (contents: 834ba9f2e7e0e9f89b1346f8).
Server, are you still there? If so, reply "HAT" (500 letters).

User Meg wants these 500 letters: HAT.

A connection. Jake requested pictures of user
requests the "missed connections" page. Eve
User (administrator) wants to set server's master
key to "14835038534". Isabel wants pages about
snakes but not too long". User Karen wants to
change account password to "C0keRaPt". User

User Meg wants these 500 letters: HAT.

HAT. Lucas requests the "missed connections" page. Eve (administrator) wants to set server's master key to "14835038534". Isabel wants pages about snakes but not too long". User Karen wants to change account password to "C0keRaPt". User

© XKCD - https://xkcd.com/1354/
void vulnerable(void *arg, size_t len) {
    long val = 0;
    long *ptr = NULL;
    char buf[128];
    ...
    memcpy(buf, arg, len);
    *ptr = val;
    ...
}

“Arbitrary write” capability: The attacker can write controlled data into a controlled location
int main(int argc, char *argv[]) {
    char *p, *q;
    p = malloc(1024);
    q = malloc(1024);
    strcpy(p, argv[1]);
    free(q);
    free(p);
    return 0;
}
Format String Vulnerabilities

The printf() family of functions accept a format string denoting how a variable will be displayed

\[
\text{printf("%s", str) } \Rightarrow \text{ prints str variable as string}
\]
\[
\text{printf("%d", num) } \Rightarrow \text{ prints num as a decimal value}
\]
\[
\text{printf("%x", num) } \Rightarrow \text{ prints num as a hexadecimal value}
\]

Format strings can also write to memory

\[
\text{printf("ABCD%n", &i) } \Rightarrow \text{ write the number of bytes output so far to the memory address of the first argument}
\]

What if...

The programmer does not supply a format string?
Fewer arguments are passed than the number of format string parameters?
Simple format string error example

```c
int main(int argc, char *argv[]) {
    printf("Input: ");
    printf(argv[1]);
    printf("\n");
}
```

$ ./fmt test
Input: test

$ ./fmt "%08x %08x %08x %08x"
input: b773c080 0804846b b7721ff4 08048460

$ ./fmt $(printf "\x18\xa0\x04\x08")%x%x%x%x%n
Safer way

```c
int main(int argc, char *argv[]) {
    printf("Input: ");
    printf("%s", argv[1]);
    printf("\n");
}
```
Other Memory-related Exploitable Conditions

NULL-termination errors
Dangling pointers
NULL pointer dereferences
String truncation
Single-byte overwrite
Off-by-one accesses
Double free

…
Race Conditions

Situations where the behavior of the program depends on the timing of some event

**Critical section**

Opens up a window of opportunity for the attacker

Race conditions occur in many different contexts

- Multi-threaded programs with different threads operating on the same data
- Distributed applications that perform multi-step transactions
- Time of check to time of use (TOCTTOU): changes may happen between *checking* a condition and *using* the results of the check

Remember the Sendmail vulnerability?
Filesystem race condition example

// setuid program

if (access("file", W_OK) != 0) {   // access() checks the
exit(1);
   // real uid (not eid)
}

fd = open("file", O_WRONLY);
write(fd, buffer, sizeof(buffer));  // write() modifies
   // /etc/passwd
iOS 8.1 Hardware-assisted Screenlock Bruteforce

Successfully brute-force device PIN even if "wipe out after 10 failed attempts" is enabled (!)

Vulnerable code:

1. Display “incorrect pin” message
2. ++attempts;

---

Correct code:

1. ++attempts;  // gets written to flash memory
2. Display “incorrect pin” message
Side Channels: TENEX Password Guessing Bug

Vulnerable password checking routine
  Check each character in succession
  Report failure on the first mismatched character

Attack: precisely align the password buffer across two pages
  Place the first password character as the last byte of the first page
  Ensure that second page is unmapped
  Try all first characters until getting a page fault → correct guess!
  Shift by one character and repeat

\[ 128^n \rightarrow 128n \]
**Program Logic Flaws: GOTO FAIL**

iOS 7.0.6 signature verification error

Legitimate-looking TLS certificates with mismatched private keys were unconditionally accepted...

```c
if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
    goto fail;
    goto fail;  // ?!!!?!!?!
if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
    goto fail;
...
fail:
    SSLFreeBuffer(&signedHashes);
    SSLFreeBuffer(&hashCtx);
    return err;
```

Check never executed