Software Security: Buffer Overflow Defenses + Misc

Fall 2021

David Kohlbrenner
dkohlbre@cs

Thanks to Franzi Roesner, Dan Boneh, Dieter Gollmann, Dan Halperin, David Kohlbrenner, Yoshi Kohno, Ada Lerner, John Manferdelli, John Mitchell, Vitaly Shmatikov, Bennet Yee, and many others for sample slides and materials ...
Admin

• Lab 1: Friday (Oct 15)
  • That is, sploits 1-3
  • When you are ‘done,’ stop changing those files.
  • Start early!
Defense: Run-Time Checking: StackGuard

- Embed "canaries" (stack cookies) in stack frames and verify their integrity prior to function return
  - Any overflow of local variables will damage the canary

- Choose random canary string on program start
  - Attacker can’t guess what the value of canary will be

- Canary contains: "\0", newline, linefeed, EOF
  - String functions like strcpy won’t copy beyond "\0"
Defeating StackGuard

• StackGuard can be defeated
  – A single memory write where the attacker controls both the value and the destination is sufficient

• Suppose program contains \texttt{copy(buf,attacker-input)} and \texttt{copy(dst,buf)}
  – Example: \texttt{dst} is a local pointer variable
  – Attacker controls both \texttt{buf} and \texttt{dst}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{diagram.png}
\caption{Diagram of StackGuard attack}
\end{figure}
ASLR: Address Space Randomization

• Randomly arrange address space of key data areas for a process
  • Base of executable region
  • Position of stack
  • Position of heap
  • Position of libraries

• Introduced by Linux PaX project in 2001
• Adopted by OpenBSD in 2003
• Adopted by Linux in 2005
ASLR: Address Space Randomization

• Deployment (examples)
  • Linux kernel since 2.6.12 (2005+)
  • Android 4.0+
  • iOS 4.3+ ; OS X 10.5+
  • Microsoft since Windows Vista (2007)

• Attacker goal: Guess or figure out target address (or addresses)

• ASLR more effective on 64-bit architectures
Attacking ASLR

• **NOP sleds** and **heap spraying** to increase likelihood for adversary’s code to be reached (e.g., on heap)

• Brute force attacks or memory disclosures to map out memory on the fly
  • Disclosing a single address can reveal the location of all code within a library, depending on the ASLR implementation
PointGuard

• Attack: overflow a function pointer so that it points to attack code

• Idea: encrypt all pointers while in memory
  • Generate a random key when program is executed
  • Each pointer is XORed with this key when loaded from memory to registers or stored back into memory
    • Pointers cannot be overflowed while in registers

• Attacker cannot predict the target program’s key
  • Even if pointer is overwritten, after XORing with key it will dereference to a “random” memory address
Normal Pointer Dereference

1. Fetch pointer value
2. Access data referenced by pointer

Memory

CPU

Pointer 0x1234

Data

0x1234

Memory

CPU

Corrupted pointer 0x1234 0x1340

Data

Attack code

0x1234 0x1340

[Cowan]
PointGuard Dereference

1. Fetch pointer value
2. Access data referenced by pointer

1. Fetch pointer value
2. Access random address; segmentation fault and crash
PointGuard Issues

• Must be very fast
  • Pointer dereferences are very common

• Compiler issues
  • Must encrypt and decrypt *only* pointers
  • If compiler “spills” registers, unencrypted pointer values end up in memory and can be overwritten there

• Attacker should not be able to modify the key
  • Store key in its own non-writable memory page

• PG’d code doesn’t mix well with normal code
  • What if PG’d code needs to pass a pointer to OS kernel?
Defense: Shadow stacks

• Idea: don’t store return addresses on the stack!

• Store them on... a different stack!
  • A hidden stack

• On function call/return
  • Store/retrieve the return address from shadow stack

• Or store on both main stack and shadow stack, and compare for equality at function return

• 2020/2021 Hardware Support emerges (e.g., Intel Tiger Lake, AMD Ryzen PRO 5000)
Challenges With Shadow Stacks

• Where do we put the shadow stack?
  • Can the attacker figure out where it is? Can they access it?

• How fast is it to store/retrieve from the shadow stack?

• How big is the shadow stack?

• Is this compatible with all software?

• (Still need to consider data corruption attacks, even if attacker can’t influence control flow.)
Other Big Classes of Defenses

• Use safe programming languages, e.g., Java, Rust
  • What about legacy C code?
  • (Though Java doesn’t magically fix all security issues 😊)

• Static analysis of source code to find overflows

• Dynamic testing: “fuzzing”
Fuzz Testing

• Generate “random” inputs to program
  • Sometimes conforming to input structures (file formats, etc.)

• See if program crashes
  • If crashes, found a bug
  • Bug may be exploitable

• Surprisingly effective

• Now standard part of development lifecycle
Other Common Software Security Issues...
Another Type of Vulnerability

• Consider this code:

```c
char buf[80];
void vulnerable() {
    int len = read_int_from_network();
    char *p = read_string_from_network();
    if (len > sizeof(buf)) {
        error("length too large, nice try!");
        return;
    }
    memcpy(buf, p, len);
}

void *memcpy(void *dst, const void * src, size_t n);
typedef unsigned int size_t;
```
Another Example

```c
size_t len = read_int_from_network();
char *buf;
buf = malloc(len+5);
read(fd, buf, len);
```

Breakout Groups

(from [www-inst.eecs.berkeley.edu—implflaws.pdf](http://www-inst.eecs.berkeley.edu—implflaws.pdf))
Implicit Cast

• Consider this code:

```c
char buf[80];
void vulnerable() {
    int len = read_int_from_network();
    char *p = read_string_from_network();
    if (len > sizeof buf) {
        error("length too large, nice try!");
        return;
    }
    memcpy(buf, p, len);
}
```

If `len` is negative, may copy huge amounts of input into `buf`.

```c
void *memcpy(void *dst, const void * src, size_t n);
typedef unsigned int size_t;
```
Integer Overflow

What if `len` is large (e.g., `len = 0xFFFFFFFF`)?

Then `len + 5 = 4` (on many platforms)

Result: Allocate a 4-byte buffer, then read a lot of data into that buffer.

(from www-inst.eecs.berkeley.edu—implflaws.pdf)
Another Type of Vulnerability

• Consider this code:

```c
if (access("file", W_OK) != 0) {
    exit(1); // user not allowed to write to file
}

fd = open("file", O_WRONLY);
write(fd, buffer, sizeof(buffer));
```

• **Goal:** Write to file only with permission

• **What can go wrong?**
TOCTOU (Race Condition)

- TOCTOU = “Time of Check to Tile of Use”

```c
if (access("file", W_OK) != 0) {
    exit(1); // user not allowed to write to file
}
fd = open("file", O_WRONLY);
write(fd, buffer, sizeof(buffer));
```

- **Goal**: Write to file only with permission
- **Attacker** (in another program) can change meaning of “file” between `access` and `open`:
  symlink("/etc/passwd", "file");
Password Checker

• Functional requirements
  • PwdCheck(RealPwd, CandidatePwd) should:
    • Return TRUE if RealPwd matches CandidatePwd
    • Return FALSE otherwise
  • RealPwd and CandidatePwd are both 8 characters long
Password Checker

• Functional requirements
  • $\text{PwdCheck}(\text{RealPwd}, \text{CandidatePwd})$ should:
    • Return TRUE if $\text{RealPwd}$ matches $\text{CandidatePwd}$
    • Return FALSE otherwise
  • $\text{RealPwd}$ and $\text{CandidatePwd}$ are both 8 characters long

• Implementation (like TENEX system)

```c
PwdCheck(RealPwd, CandidatePwd) // both 8 chars
for i = 1 to 8 do
    if (RealPwd[i] != CandidatePwd[i]) then
        return FALSE
return TRUE
```

• Clearly meets functional description
Attacker Model

- Attacker can guess CandidatePwds through some standard interface
- Naive: Try all $256^8 = 18,446,744,073,709,551,616$ possibilities
- Is it possible to derive password more quickly?

```
PwdCheck(RealPwd, CandidatePwd)  // both 8 chars
    for i = 1 to 8 do
        if (RealPwd[i] != CandidatePwd[i]) then
            return FALSE
    return TRUE
```