Software Security: Buffer Overflow Defenses and Miscellaneous

Spring 2017

Franziska (Franzi) Roesner
franzi@cs.washington.edu

Thanks to Dan Boneh, Dieter Gollmann, Dan Halperin, Yoshi Kohno, Ada Lerner, John Manferdelli, John Mitchell, Vitaly Shmatikov, Bennet Yee, and many others for sample slides and materials ...
Looking Forward

• **Today:** more on software security
• **Friday:** guest lecture by Karl Koscher
• **Next week:** finish software security, start crypto

• **Reading #1** due Thursday (584M only)
• **Homework #1** due Friday
• **Lab #1** out!
  – Submit your group + public key to the form sent out via email
  – Instructions for creating a key are in the lab description

• **Section this week:** Lab 1
Buffer Overflow: Causes and Cures

• Typical memory exploit involves code injection
  – Put malicious code at a predictable location in memory, usually masquerading as data
  – Trick vulnerable program into passing control to it

• Possible defenses:
  1. Prevent execution of untrusted code
  2. Stack “canaries”
  3. Encrypt pointers
  4. Address space layout randomization
**W-xor-X / DEP**

- Mark all writeable memory locations as non-executable
  - Example: Microsoft’s Data Execution Prevention (DEP)
  - This blocks (almost) all code injection exploits
- Hardware support
  - AMD “NX” bit, Intel “XD” bit (in post-2004 CPUs)
  - Makes memory page non-executable
- Widely deployed
  - Windows (since XP SP2), Linux (via PaX patches), OS X (since 10.5)
What Does W-xor-X Not Prevent?

• Can still corrupt stack …
  – … or function pointers or critical data on the heap
• As long as “saved EIP” points into existing code, W-xor-X protection will not block control transfer
• This is the basis of return-to-libc exploits
  – Overwrite saved EIP with address of any library routine, arrange stack to look like arguments
• Does not look like a huge threat
  – Attacker cannot execute arbitrary code
return-to-libc on Steroids

• Overwritten saved EIP need not point to the beginning of a library routine

• **Any** existing instruction in the code image is fine
  – Will execute the sequence starting from this instruction

• What if instruction sequence contains RET?
  – Execution will be transferred... to where?
  – Read the word pointed to by stack pointer (ESP)
    • Guess what? Its value is under attacker’s control!
  – Use it as the new value for EIP
    • Now control is transferred to an address of attacker’s choice!
  – Increment ESP to point to the next word on the stack
Chaining RETs for Fun and Profit

• Can chain together sequences ending in RET
  – Krahmer, “x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique” (2005)

• What is this good for?

• Answer [Shacham et al.]: everything
  – Turing-complete language
  – Build “gadgets” for load-store, arithmetic, logic, control flow, system calls
  – Attack can perform arbitrary computation using no injected code at all – return-oriented programming
Return-Oriented Programming
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

![Diagram of stack frame with canary](image)
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

• Terminator canary: “\0”, newline, linefeed, EOF
  – String functions like strcpy won’t copy beyond “\0”
StackGuard Implementation

- StackGuard requires code recompilation
- Checking canary integrity prior to every function return causes a performance penalty
  - For example, 8% for Apache Web server
- StackGuard can be defeated
  - A single memory write where the attacker controls both the value and the destination is sufficient
Defeating StackGuard

- Suppose program contains `strcpy(dst,buf)` where attacker controls both `dst` and `buf`
  - Example: `dst` is a local pointer variable

![Diagram of StackGuard bypass](image)

- Overwrite destination of `strcpy` with RET position
  - `strcpy` will copy `BadPointer` here

- BadPointer, attack code
  - Overwrite destination of `strcpy` with RET position
  - Return execution to this address
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

CPU

Memory

Pointer 0x1234
Data
0x1234

CPU

Memory

Corrupted pointer 0x1234 0x1340
Data
Attack code
0x1234 0x1340

1. Fetch pointer value
2. Access attack code referenced by corrupted pointer

[Cowan]
PointGuard Dereference

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

Memory

CPU

Decrypt

0x1234

Encrypted pointer
0x7239

Data
0x1234

CPU

Decrypts to random value

0x9786

Memory

Corrupted pointer
0x7239
0x1340

Data
0x1234

Attack code
0x1340

0x9786

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?
ASLR: Address Space Randomization

• Map shared libraries to a random location in process memory
  – Attacker does not know addresses of executable code

• Deployment (examples)
  – Windows Vista: 8 bits of randomness for DLLs
  – Linux (via PaX): 16 bits of randomness for libraries
  – Even Android
  – More effective on 64-bit architectures

• Other randomization methods
  – Randomize system call ids or instruction set
Example: ASLR in Vista

• Booting Vista twice loads libraries into different locations:

<table>
<thead>
<tr>
<th>File</th>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ntlanman.dll</td>
<td>0x6D7F0000</td>
<td>Microsoft® Lan Manager</td>
</tr>
<tr>
<td>ntmarta.dll</td>
<td>0x75370000</td>
<td>Windows NT MARTA provider</td>
</tr>
<tr>
<td>ntshrui.dll</td>
<td>0x6F2C0000</td>
<td>Shell extensions for sharing</td>
</tr>
<tr>
<td>ole32.dll</td>
<td>0x76160000</td>
<td>Microsoft OLE for Windows</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>File</th>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ntlanman.dll</td>
<td>0x6DA90000</td>
<td>Microsoft® Lan Manager</td>
</tr>
<tr>
<td>ntmarta.dll</td>
<td>0x75660000</td>
<td>Windows NT MARTA provider</td>
</tr>
<tr>
<td>ntshrui.dll</td>
<td>0x6D9D0000</td>
<td>Shell extensions for sharing</td>
</tr>
<tr>
<td>ole32.dll</td>
<td>0x763C0000</td>
<td>Microsoft OLE for Windows</td>
</tr>
</tbody>
</table>
ASLR Issues

• NOP slides and heap spraying to increase likelihood for custom code (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
Other Possible Solutions

• Use safe programming languages, e.g., Java
  – 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”
• LibSafe: dynamically loaded library that intercepts calls to unsafe C functions and checks that there’s enough space before doing copies
  – Also doesn’t prevent everything
Beyond Buffer Overflows...
Another Type of Vulnerability

• Consider this code:

```c
int openfile(char *path) {
    struct stat s;
    if (stat(path, &s) < 0)
        return -1;
    if (!S_ISREG(s.st_mode)) {
        error("only allowed to regular files!");
        return -1;
    }
    return open(path, O_RDONLY);
}
```

• **Goal:** Open only regular files (not symlink, etc)
• **What can go wrong?**
TOCTOU (Race Condition)

• TOCTOU == Time of Check to Time of Use:

```c
int openfile(char *path) {
    struct stat s;
    if (stat(path, &s) < 0)
        return -1;
    if (!S_ISREG(s.st_mode)) {
        error("only allowed to regular files!");
        return -1;
    }
    return open(path, O_RDONLY);
}
```

• **Goal:** Open only regular files (not symlink, etc)
• Attacker can change meaning of `path` between `stat` and `open` (and access files he or she shouldn’t)
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);
}
```

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

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

(from www-inst.eecs.berkeley.edu—implflaws.pdf)
• 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](http://www-inst.eecs.berkeley.edu/~implflaws.pdf))
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

• 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
- Better: Time how long it takes to reject a CandidatePasswd. Then try all possibilities for first character, then second, then third, ....
  - Total tries: $256^8 = 2048$

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

• Assume there are no “typical” bugs in the software
  – No buffer overflow bugs
  – No format string vulnerabilities
  – Good choice of randomness
  – Good design

• The software may still be vulnerable to timing attacks
  – Software exhibits input-dependent timings

• Complex and hard to fully protect against
Other Examples

• Plenty of other examples of timings attacks
  – AES cache misses
    • AES is the “Advanced Encryption Standard”
    • It is used in SSH, SSL, IPsec, PGP, ...
  – RSA exponentiation time
    • RSA is a famous public-key encryption scheme
    • It’s also used in many cryptographic protocols and products