Software Security (II):
Other types of software vulnerabilities
#293 HRE-THR 850 1930
ALICE SMITH
COACH

SPECIAL INSTRUX:
NONE
To comply with the TSA Secure Flight program, the traveler information listed here must exactly match the information on the government-issued photo ID that the traveler presents at the airport.

Title (optional): Dr.  First Name: Alice  Middle Name:  Last Name: Smith

Gender: Female  Date of Birth: 01/24/93

Travelers are required to enter a middle name/initial if one is listed on their government-issued photo ID.

Some younger travelers are not required to present an ID when traveling within the U.S. Learn more

Known Traveler Number/Pass ID (optional): ?

Redress Number (optional): ?

Seat Request:
- No Preference
- Aisle
- Window
#293 HRE-THR 850 1930
ALICE SMITHHHHHHHHHHHH
HHACH

SPECIAL INSTRUX: NONE
Traveler 1 - Adults (age 18 to 64)

To comply with the TSA Secure Flight program, the traveler information listed here must exactly match the information on the government-issued photo ID that the traveler presents at the airport.

Title (optional): Dr.  First Name: Alice  Middle Name:  Last Name: Smith  First

Gender: Female  Date of Birth: 01/24/93

Travelers are required to enter a middle name/initial if one is listed on their government-issued photo ID.

Some younger travelers are not required to present an ID when traveling within the U.S. Learn more

Known Traveler Number/Pass ID (optional):  Redress Number (optional): 

Seat Request:  No Preference  Aisle  Window
#293 HRE-THR 850 1930
ALICE SMITH
FIRST

SPECIAL INSTRUX:
NONE
Example #1

```c
void vulnerable() {
    char name[20];
    ...
    gets(name);
    ...
}
```
Example #2

```c
void vulnerable() {
    char instrux[80] = "none";
    char name[20];
    ...
    gets(name);
    ...
}
```
void vulnerable() {
    char cmd[80];
    char line[512];
    ...
    strncpy(cmd,"/usr/bin/finger", 80);
    gets(line);
    ...
    execv(cmd, ...);
}
Example #4

```c
void vulnerable() {
    int (*fnptr)();
    char buf[80];
    ...
    gets(buf);
    ...
}
```
void vulnerable() {
    int seatInFirstClass = 0;
    char name[20];
    ...
    gets(name);
    ...
}
Example #6

```c
void vulnerable() {
    int authenticated = 0;
    char name[20];

    ...

    gets(name);

    ...
}
```
Common Coding Errors

• Input validation vulnerabilities

• Memory management vulnerabilities

• TOCTTOU vulnerability (later)
Input validation vulnerabilities

• Program requires certain assumptions on inputs to run properly
• Without correct checking for inputs
  – Program gets exploited
• Example:
  – Buffer overflow
  – Format string
1: unsigned int size;
2: Data **datalist;
3: 
4: size = GetUntrustedSizeValue();
5: datalist = (data **)malloc(size * sizeof(Data *));
6: for(int i=0; i<size; i++) {
7:     datalist[i] = InitData();
8: } 
9: datalist[size] = NULL;
10: ...
Example II

• What's wrong with this code?
• Hint – \texttt{memcpy()} prototype:
  – \texttt{void *memcpy(void *dest, const void *src, size_t n)};
• Definition of \texttt{size_t}: typedef unsigned int size_t;
• Do you see it now?
Implicit Casting Bug

• Attacker provides a negative value for len
  – if won’t notice anything wrong
  – Execute `memcpy()` with negative third arg
  – Third arg is implicitly cast to an unsigned int, and becomes a very large positive int
  – `memcpy()` copies huge amount of memory into `buf`, yielding a buffer overrun!

• A signed/unsigned or an implicit casting bug
  – Very nasty – hard to spot

• C compiler doesn’t warn about type mismatch between signed int and unsigned int
  – Silently inserts an implicit cast
Example III (Integer Overflow)

What’s wrong with this code?
- No buffer overrun problems (5 spare bytes)
- No sign problems (all ints are unsigned)

But, len+5 can overflow if len is too large
- If len = 0xFFFFFFFF, then len+5 is 4
- Allocate 4-byte buffer then read a lot more than 4 bytes into it: classic buffer overrun!

Know programming language’s semantics well to avoid pitfalls
Example IV

Example IV

```
1:  char* ptr = (char*) malloc(SIZE);
2:  if (err) {
3:    abrt = 1;
4:    free(ptr);
5:  }
6:  ...
7:  if (abrt) {
8:    logError(“operation aborted before commit”, ptr);
9:  }
```

- Use-after-free
- Corrupt memory
Example V

1: char* ptr = (char*) malloc(SIZE);
2: if (err) {
3:    abrt = 1;
4:    free(ptr);
5: }
6: ...
7: free(ptr);

• Double-free error
• Corrupts memory-management data structure
Example VI: Format string problem

Example VI

```c
int func(char *user) {
    fprintf(stderr, user);
}
```
Format Functions

• Used to convert simple C data types to a string representation
• Variable number of arguments
• Including format string
• Example
  – printf(“%s number %d”, “block”, 2)
  – Output: “block number 2”
## Format String Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Output</th>
<th>Passed as</th>
</tr>
</thead>
<tbody>
<tr>
<td>%d</td>
<td>Decimal (int)</td>
<td>Value</td>
</tr>
<tr>
<td>%u</td>
<td>Unsigned decimal (unsigned int)</td>
<td>Value</td>
</tr>
<tr>
<td>%x</td>
<td>Hexadecimal (unsigned int)</td>
<td>Value</td>
</tr>
<tr>
<td>%s</td>
<td>String ((const) (unsigned) char *)</td>
<td>Reference</td>
</tr>
<tr>
<td>%n</td>
<td># bytes written so far, (* int)</td>
<td>Reference</td>
</tr>
</tbody>
</table>
Example VI: Format string problem

Example VI

```c
int func(char *user) {
    fprintf(stderr, user);
}
```

- **Problem:** what if `*user = "%s%s%s%s%s%s%s%s` ??
  - `%s` displays memory
  - Likely to read from an illegal address
  - If not, program will print memory contents.

**Correct form:**

```c
fprintf(stdout, "%s", user);
```
Stack and Format Strings

- Function behavior is controlled by the format string
- Retrieves parameters from stack as requested: “%”
- Example:

```c
printf("Number %d has no address, number %d has: %08x\n", I, a, &a)
```

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Address of the format string</td>
</tr>
<tr>
<td>i</td>
<td>Value of variable I</td>
</tr>
<tr>
<td>a</td>
<td>Value of variable a</td>
</tr>
<tr>
<td>&amp;a</td>
<td>Address of variable a</td>
</tr>
</tbody>
</table>
View Stack

• `printf("%08x. %08x. %08x. %08x\n")`
  – 40012983.0806ba43.bfffff4a.0802738b

• display 4 values from stack
Read Arbitrary Memory

- char input[] = “\x10\x01\x48\x08_%08x. %08x. %08x. %08x|%s|”;  
  printf(input)
  - Will display memory from 0x08480110

- Uses reads to move stack pointer into format string

- %s will read at 0x08480110 till it reaches null byte
Writing to arbitrary memory

- `printf( "hello \%n", &temp)`
  - writes ‘6’ into temp.

- `printf( "\%08x.\%08x.\%08x.\%08x.\%n")`
Vulnerable functions

Any function using a format string.

Printing:
  printf, fprintf, sprintf, ...
  vprintf, vfprintf, vsprintf, ...

Logging:
  syslog, err, warn
An Exploit Example

```c
syslog("Reading username:");
read_socket(username);
syslog(username);
```

Welcome to InsecureCorp. Please login.
Login: EvilUser%s%s...%400n...%n
root@server> _
Why The Bug Exists

• C language has poor support for variable-argument functions
  – Callee doesn’t know the number of actual args
• No run-time checking for consistency between format string and other args
• Programmer error
Real-world Vulnerability Samples

• First exploit discovered in June 2000.
• Examples:
  – wu-ftpd  2.*: remote root
  – Linux rpc.statd: remote root
  – IRIX telnetd: remote root
  – BSD chpass: local root
What are software vulnerabilities?

• Flaws in software
• Break certain assumptions important for security
  – E.g., what assumptions are broken in buffer overflow?
Why does software have vulnerabilities?

• Programmers are humans!
  – Humans make mistakes!

• Programmers are not security-aware

• Programming languages are not designed well for security
What can you do?

• Programmers are humans!
  – Humans make mistakes!
  – Use tools! (next lecture)

• Programmers were not security aware
  – Learn about different common classes of coding errors

• Programming languages are not designed well for security
  – Pick better languages
Software Security (III):
Defenses against Memory-Safety Exploits
Preventing hijacking attacks

**Fix bugs:**
- Audit software
  - Automated tools: Coverity, Prefast/Prefix, Fortify
- Rewrite software in a type-safe language (Java, ML)
  - Difficult for existing (legacy) code ...

**Allow overflow, but prevent code execution**

**Add runtime code to detect overflows exploits:**
- Halt process when overflow exploit detected
- StackGuard, Libsafe
## Control-hijacking Attack Space

<table>
<thead>
<tr>
<th>Defenses/Mitigations</th>
<th>Code Injection</th>
<th>Arc Injection</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Heap</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Exception Handlers</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Defense I: non-execute \((W^X)\)

Prevent attack code execution by marking stack and heap as **non-executable**

- **NX-bit** on AMD Athlon 64, **XD-bit** on Intel P4 Prescott
  - NX bit in every Page Table Entry (PTE)

- **Deployment:**
  - Linux (via PaX project); OpenBSD
  - Windows: since XP SP2 (DEP)
    - Boot.ini: /noexecute=OptIn or **AlwaysOn**
    - Visual Studio: /NXCompat[:NO]
Effectiveness and Limitations

- **Limitations:**
  - Some apps need executable heap (e.g. JITs).
  - Does not defend against exploits using return-oriented programming (ROP).

<table>
<thead>
<tr>
<th></th>
<th>Code Injection</th>
<th>Arc Injection</th>
<th>Non-Execute (NX)*</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack</td>
<td>Non-Execute (NX)*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Heap</td>
<td>Non-Execute (NX)*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Exception Handlers</td>
<td>Non-Execute (NX)*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Handlers</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

* When Applicable
Return-Oriented Programming (ROP)

- ret2lib exploits
  - Reuse existing functions, no code injection required
Ret-2-lib Exploit

So suppose we want to spawn a shell by exploiting a buffer overflow vulnerability:

**Shell Code:** `system(“/bin/sh”)`

When the function exits, it returns to the entry of the libc function `system`. With the crafted argument, the user gets a shell !!!
Return-Oriented Programming (ROP)

• **ret2lib exploits**
  – Reuse existing functions, no code injection required

• **Return-oriented programming**
  – Reuses existing code chunks (called gadgets)
  – The gadgets could provide a Turing-complete exploit language

Buchanan et. al, BlackHat 2008
**Defense II: Address Randomization**

**ASLR:** (Address Space Layout Randomization)
- Start stack at a random location
- Start heap at a random location
- Map shared libraries to random location in process memory
  ⇒ Attacker cannot jump directly to exec function

**Deployment:** (/DynamicBase)
- **Windows** Vista: 8 bits of randomness for DLLs
  - aligned to 64K page in a 16MB region ⇒ 256 choices
- **Linux** (via PaX): 16 bits of randomness for libraries
- More effective on 64-bit architectures

**Other randomization methods:**
- Sys-call randomization: randomize sys-call id’s
- Instruction Set Randomization (ISR)
Effectiveness and Limitations

- Limitations
  - Randomness is limited
  - Some vulnerabilities can allow secret to be leaked

<table>
<thead>
<tr>
<th>Defense/Mitigations</th>
<th>Code Injection</th>
<th>Arc Injection</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack</td>
<td>Non-Execute (NX)* ASLR</td>
<td>ASLR</td>
</tr>
<tr>
<td>Heap</td>
<td>Non-Execute (NX)* ASLR</td>
<td>ASLR</td>
</tr>
<tr>
<td>Exception Handlers</td>
<td>Non-Execute (NX)* ASLR</td>
<td>ASLR</td>
</tr>
</tbody>
</table>

* When Applicable
Defense III: StackGuard

• Run time tests for stack integrity

• Embed “canaries” in stack frames and verify their integrity prior to function return
Canary Types

• **Random canary:**
  – Random string chosen at program startup.
  – Insert canary string into every stack frame.
  – Verify canary before returning from function.
    • Exit program if canary changed. Turns potential exploit into DoS.
  – To exploit successfully, attacker must learn current random string.

• **Terminator canary:**  
  Canary =  \{0, newline, linefeed, EOF\}
  – String functions will not copy beyond terminator.
  – Attacker cannot use string functions to corrupt stack.
StackGuard (Cont.)

• StackGuard implemented as a GCC patch.
  – Program must be recompiled.

• Low performance effects: 8% for Apache.

• Note: Canaries don’t provide full proof protection.
  – Some stack smashing attacks leave canaries unchanged

• Heap protection: PointGuard.
  – Protects function pointers and setjmp buffers by encrypting them: e.g. XOR with random cookie
  – Less effective, more noticeable performance effects
StackGuard enhancements: ProPolice

- ProPolice (IBM) - gcc 3.4.1. (-fstack-protector)
  - Rearrange stack layout to prevent ptr overflow.

- String Growth
- Stack Growth

- arguments
- return address
- stack frame pointer
- CANARY
- local string buffers
- local string variables
- local non-buffer variables
- copy of pointer args

Protects pointer args and local pointers from a buffer overflow.

pointers, but no arrays

Dawn Song
MS Visual Studio /GS [since 2003]

Compiler /GS option:
- Combination of ProPolice and Random canary.
- If cookie mismatch, default behavior is to call _exit(3)

Function prolog:
- sub esp, 8 // allocate 8 bytes for cookie
- mov eax, DWORD PTR __security_cookie
- xor eax, esp // xor cookie with current esp
- mov DWORD PTR [esp+8], eax // save in stack

Function epilog:
- mov ecx, DWORD PTR [esp+8]
- xor ecx, esp
- call @__security_check_cookie
- add esp, 8

Enhanced /GS in Visual Studio 2010:
- /GS protection added to all functions, unless can be proven unnecessary

Dawn Song
/GS stack frame

String Growth

arguments
return address
stack frame pointer
exception handlers
CANARY
local string buffers
local string variables
local non-buffer variables

Stack Growth

Canary protects ret-addr and exception handler frame

pointers, but no arrays

Dawn Song
## Effectiveness and Limitations

- **Limitation:**
  - Evasion with exception handler* When Applicable

<table>
<thead>
<tr>
<th></th>
<th>Code Injection</th>
<th>Arc Injection</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Stack</strong></td>
<td>Non-Execute (NX)* ASLR</td>
<td>ASLR StackGuard(Canaries) ProPolice/GS</td>
</tr>
<tr>
<td></td>
<td>StackGuard(Canaries) ProPolice/GS</td>
<td></td>
</tr>
<tr>
<td><strong>Heap</strong></td>
<td>Non-Execute (NX)* ASLR PointGuard</td>
<td>ASLR PointGuard</td>
</tr>
<tr>
<td><strong>Exception Handlers</strong></td>
<td>Non-Execute (NX)* ASLR</td>
<td>ASLR</td>
</tr>
</tbody>
</table>

* When Applicable
Evading /GS with exception handlers

• When exception is thrown, dispatcher walks up exception list until handler is found (else use default handler)

After overflow: handler points to attacker’s code
exception triggered ⇒ control hijack
Main point: exception is triggered before canary is checked
Defense III: SAFESEH and SEHOP

- **SAFESEH**: linker flag
  - Linker produces a binary with a table of safe exception handlers
  - System will not jump to exception handler not on list

- **SEHOP**: platform defense (since win vista SP1)
  - Observation: SEH attacks typically corrupt the “next” entry in SEH list.
  - SEHOP: add a dummy record at top of SEH list
  - When exception occurs, dispatcher walks up list and verifies dummy record is there. If not, terminates process.
## Effectiveness and Limitations

- **Limitations:**
  - Require recompilation

<table>
<thead>
<tr>
<th>Defenses/Mitigations</th>
<th>Code Injection</th>
<th>Arc Injection</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Stack</strong></td>
<td>Non-Execute (NX)*, ASLR, StacKGuard(Canaries), ProPolice /GS</td>
<td>ASLR, StacKGuard(Canaries), ProPolice /GS</td>
</tr>
<tr>
<td><strong>Heap</strong></td>
<td>Non-Execute (NX)*, ASLR, PointGuard</td>
<td>ASLR, PointGuard</td>
</tr>
<tr>
<td><strong>Exception Handlers</strong></td>
<td>Non-Execute (NX)*, ASLR, <strong>SAFESEH and SEHOP</strong></td>
<td>ASLR, <strong>SAFESEH and SEHOP</strong></td>
</tr>
</tbody>
</table>

* When Applicable
Defense IV: Libsafe

• Dynamically loaded library
  (no need to recompile app.)
• Intercepts calls to `strcpy(dest, src)`
  – Validates sufficient space in current stack frame:
    
    \[|\text{frame-pointer} - \text{dest}| > \text{strlen(src)}\]
  – If so, does `strcpy`. Otherwise, terminates application
# Effectiveness and Limitations

## Limitations:

- Limited protection

<table>
<thead>
<tr>
<th></th>
<th>Code Injection</th>
<th>Arc Injection</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Stack</strong></td>
<td>Non-Execute (NX)* ASLR StackGuard(Canaries) ProPolice /GS libsafe</td>
<td>ASLR StackGuard(Canaries) ProPolice /GS libsafe</td>
</tr>
<tr>
<td><strong>Heap</strong></td>
<td>Non-Execute (NX)* ASLR PointGuard</td>
<td>ASLR PointGuard</td>
</tr>
<tr>
<td><strong>Exception Handlers</strong></td>
<td>Non-Execute (NX)* ASLR SAFESEH and SEHOP</td>
<td>ASLR SAFESEH and SEHOP</td>
</tr>
</tbody>
</table>

*When Applicable*
Other Defenses

- **StackShield**
  - At function prologue, copy return address RET and SFP to “safe” location (beginning of data segment)
  - Upon return, check that RET and SFP is equal to copy.
  - Implemented as assembler file processor (GCC)

- **Control Flow Integrity** (CFI)
  - A combination of static and dynamic checking
    - Statically determine program control flow
    - Dynamically enforce control flow integrity
# Effectiveness and Limitations

- Many different kinds of attacks. Not one silver bullet defense.

<table>
<thead>
<tr>
<th>Defenses/Mitigations</th>
<th>Code Injection</th>
<th>Arc Injection</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Stack</strong></td>
<td>Non-Execute (NX)*</td>
<td>ASLR</td>
</tr>
<tr>
<td></td>
<td>ASLR</td>
<td>StackGuard(Canaries)</td>
</tr>
<tr>
<td></td>
<td>StackGuard(Canaries)</td>
<td>ProPolice</td>
</tr>
<tr>
<td></td>
<td>ProPolice</td>
<td>/GS libsafe</td>
</tr>
<tr>
<td></td>
<td>/GS libsafe</td>
<td><strong>StackShield</strong></td>
</tr>
<tr>
<td><strong>Heap</strong></td>
<td>Non-Execute (NX)*</td>
<td>ASLR</td>
</tr>
<tr>
<td></td>
<td>ASLR</td>
<td>PointGuard</td>
</tr>
<tr>
<td><strong>Exception Handlers</strong></td>
<td>Non-Execute (NX)*</td>
<td>ASLR</td>
</tr>
<tr>
<td></td>
<td>ASLR</td>
<td>SAFESEH and SEHOP</td>
</tr>
</tbody>
</table>

* When Applicable