Defending Buffer Overflow

Wei Wang
The Buffer Overflow So Far

- The buffer overflow vulnerability
- Attacks that simply crashes a program
- Code Injection – inject code into stack to execute
  - Return to original program to avoid checks
  - Return to libc
  - ROP (Return Oriented Programming)
- Heap/Data section buffer overflow
- Changing local variables and pointer subterfuge
The Requirements of Buffer Overflow Attack

• Basic requirement: an unprotected buffer
  – Crashing only requires an unprotected buffer

• Code Injection
  – Stack Address is known
  – Executable Stack

• Arc Injection
  – Addresses of useful instructions or functions must be known

• Local Var Corruption and Pointer Subterfuge
  – Buffer overrun can change important variables
The Requirements of Buffer Overflow Attack

- Heap buffer overflow
  - free() function does not check fd/bk pointers
- Because the attackers needs these requirements to launch an attack, our defense focuses on breaking these requirements
Secure Programming – Eliminate Unprotected Buffers

- Good coding style
  - Use only the good form of printf(); never use printf(buffer) for any function in the printf family
  - Review loop bounds for off-by-one errors
  - Avoid unsafe C functions (e.g. strcpy(), strcat(), sprintf(), gets(), scanf()) and learn how to use alternatives (e.g. strncpy(), strncat(), snprintf())
  - Insert bounds checking code
  - Avoid unsafe programming languages (C, C++) and use more modern, safe languages wherever possible (Java, Ada, C# in managed mode)
- Code Review
Safe and Unsafe Coding

• Unsafe

```c
void main() {
    char buf[1024];
    gets(buf);  /* Won’t stop at 1024 bytes !! */
}
```

• Safe

```c
void main() {
    char buf[1024];
    fgets(buf, 1024, stdin);  /* stop at 1024 bytes */
}
```
Safe and Unsafe Coding cont'd

- Unsafe
  
  ```c
  strcpy(dst, src);
  ```

- Safe
  
  ```c
  strncpy(dst, src, sizeof(dst)-1); /* What prevents buffer overflow? */
  dst[sizeof(dst)-1] = '\0'; /* Null terminate */
  ```
Safe and Unsafe Coding cont'd

- Unsafe

```
sprintf(dst, "%s", src);
```

- Safe

```
snprintf(dst, sizeof(dst), "%s", src); /* What prevents buffer overflow? */
```
However, Buffer Overflow Bugs are Hard to prevent

- Buffer overflow bugs can be the result of program complexity:
  - `MultiByteToWideChar` converts ASCII `username` to UTF16 wide char `unicodeUser`. But the size of `unicodeUser` is incorrect as the “size in bytes” is passed in.
  ```c
  void getUserInfo(char *username, struct _USER_INFO_2 info){
      WCHAR unicodeUser[UNLEN+1];
      MultiByteToWideChar(CP_ACP, 0, username, -1,
                          unicodeUser,
                          sizeof(unicodeUser));
  }
  ```
- Another common error is incorrect pointer arithmetic
- There are still many buffer overflow bugs in CVE every year
- Because it is impossible to eliminate buffer overflow, compiler-, runtime- and OS-based solutions are also proposed
Compiler-Based Defense – Static Analysis and Modified Language

- Compiler-based defense usually aims at preventing or detecting buffer overflow
- GCC warns about using unsafe functions
  - e.g., "warning: 'gets' is deprecated"
- Automatic bounds checking – compilers generate code to check pointers at calculation time or dereference time
  - to ensure pointers are not cross boundary
  - -fbounds-check in GCC (Jones and Kelly, 1995)
  - Fail-Safe C (Oiwa, 2009, PLDI), modified C with run-time help
  - Address Sanitizer (Serebryany et al. 2012 USENIX ATC)
- Not 100% protection
- High overhead
Compiler-Based Defense – Stack Guard: Stack Canary

• Cowan et al., USENIX Security Symp. 1997

• StackGuard inserts a marker in between the frame pointer and the return address on the stack
  – Marker is called a canary, as in the “canary in the coal mine”

• If a buffer overflow overwrites the stack all the way to the return address, it will also overwrite the canary

• Before returning, the canary is examined for modification
Compiler-Based Defense – Stack Guard: Stack Canary cont’d

• Assembly code example

```assembly
# func prologue
movl  %gs:20, %eax
pushl %eax
pushl %ebp
movl  %esp, %ebp

# func epilogue
leave
popl  %eax
xorl  %eax, %gs:20
je    .finish
call  __stack_chk_fail
.finish:
ret
```

Random value inserted between buffer and saved EBP

Random value is checked; if the value is changed, raise an exception

Return Address
Stack Canary
Saved EBP
Local Variables
Buffer
Compiler-Based Defense – Stack Guard: Stack Canary cont'd

- Defeating stack guard (and shortcomings):
  - Stack guard does not protect EBP
    - As we learned in the off-by-one example, unprotected EBP can lead to serious consequence
  - Stack guard does not protect local variables
    - Recall pointer subterfuge
  - Performance overhead
    - Requires memory access and checks every time function is called. E.g., 8% for Apache web servers.
Compiler-Based Defense – Stack Smash Protector

- Stack Smash Protector, SSP: a.k.a ProPolice
- Developed from 2001-2005, originally from IBM
- Better Canary
  - Canary is put under saved EBP
- Better memory layout
  - Local variables are put below buffers/arrays
Compiler-Based Defense – Stack Smash Protector cont'd

• Assembly code example

```
# func prologue
pushl %ebp
movl %esp, %ebp
movl %gs:20, %eax
pushl %eax
# allocate local vars
# allocate arrays

# func epilogue
popl %eax
xorl %eax, %gs:20
je .finish
call __stack_chk_fail
.finish:
leave
ret
```

Return Address
Saved EBP
Stack Canary
Buffer/Array
Local Variables

Random value inserted between buffer and saved EBP

Random value is checked; if the value is changed, raise an exception

Return Address
Saved EBP
Stack Canary
Buffer/Array
Local Variables
Compiler-Based Defense – Stack Smash Protector cont'd

- Defeating SSP (and shortcomings):
  - Does not protect against certain unsafe pointer arithmetic errors (CVE-2008-0226)
  - Exception handler address is unprotected
    - David Litchfield 2003, targeted Windows 2003 Server
    - Address of exception handler is saved on stack
    - Buffer overflow corrupted both canary and exception handler address
    - When canary check failed, bogus exception handler is called
  - Windows provided better defense in Server 2008
Compiler-Based Defense – Stack Smash Protector cont'd

- More on Defeating SSP (and shortcomings):
  - Stack canary is only checked when function returns, a lot of things can happen before return
    - E.g., function foo is protected with canary, and has an unprotected buffer
    - Buffer is overrun and v-table is corrupted
    - Any virtual functions called by foo becomes malicious
    - Only when foo returns the overflow is detected
Compiler-Based Defense – Stack Smash Protector cont'd

• More on SSP's shortcoming:
  – Performance overhead
  – No heap overflow protection

• GCC implementation:
  – -fstack-protector
    • Protects functions with buffers > 8 bytes, and “alloca”-calling functions
  – -fstack-protector-all
    • All functions are protected, better protection
    • High performance overhead
  – -fstack-protector-strong
    • A compromise between protection-range and performance
Runtime-based Defense

• Runtime library-based defense aims at providing runtime boundary checks, pointer checks or control flow checks

• libsafe (Baratloo et al. USENIX ATC 2000)
  – Intercept unsafe lib C functions such as gets, strcpy, and execute libsafe's version of gets and strcpy etc.
  – libsafe's gets or strcpy etc. checks the size of dest and src to make sure saved EBP and return addresses will not be written
Runtime-based Defense cont'd

- Defeating Libsafe (and shortcomings):
  - Does not protect corruption of local variables – because libsafe does not know actual buffer size
  - Not all buffer overflows caused by unsafe libc functions
  - It is hard to cover all unsafe functions
  - Overhead
Runtime-based Defense cont'd

• GNU libc heap protection
  – Check pointer consistence when calling `unlink`
  – This protection is designed against the heap overflow attack that corrupt `malloc` meta data

• There has been a myriad of compiler- or run-time based protections
  – e.g., function pointer encryption; return address relocation etc; control-flow protection against ROP
  – Most provide protection against certain types of attacks
  – Most have high overhead
Operating System-based Defense

- OS-based Defense aims at defeating the requirements of
  - Code execution in unsafe memory
  - Knowing the stack/heap addresses and instruction addresses

- Data Execution Prevention
Operating System-based Defense: Data Execution Prevention

- Data Execution Prevention (DEP) or Execution Space Protection or No-eXecute (NX) bit or “W xor X”
  - All basically one thing: do not allow execution of instructions on stack or heap
  - Supported by processor (since AMD64): each memory page is added with privilege – readable, write-able, and executable. If NX bit is set, the page is not executable
    - Hardware support reduces overhead
  - “W xor X” principle: a page is either writable or executable, never both; enforced at OS-level
    - Initially software techniques, high overhead; later w/. HW support
Operating System-based Defense: Data Execution Prevention cont'd

• Defeating DEP:
  – Arc-injection and ROP are designed to defeat DEP because they do not need code injection
Operating System-based Defense: ASLR

- However, arc-injection and ROP require knowing the addresses of return addresses, saved EBP and useful instructions
- Address Space Layout Randomization (ASLR) defeats that
  - Random padding to the beginning of stack
  - Random padding to the beginning of heap
  - Random beginning address of memory/file mappings
    - Libraries are loaded as file mappings (compiled as PIC, position-independent-code)
Operating System-based Defense: ASLR cont'd

- Defeating ASLR (and its shortcomings):
  - On 32-bit systems (and Macs?), the random padding does not have enough entropy (Shacham et al. CCS'04)
    - i.e. the size of the padding can be guessed by repeatedly attacking
  - Application code pages are not always randomized
    - Applications are not compiled with PIE (position independent executable), so not randomizable
    - Makes it possible to do ROP attack
    - Linux PaX has a version that randomize program code without PIE, but too slow
  - Increase executable size
  - Overhead on 32-bit OS

- Linux's PaX implements both DEP and ASLR
<table>
<thead>
<tr>
<th>Protection</th>
<th>Compiler</th>
<th>Operating System</th>
<th>ASLR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack Smash</td>
<td>SSP</td>
<td>PIC/PIE</td>
<td>DEP</td>
</tr>
<tr>
<td>corrupt ret addr and saved EBP</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Unsafe Pointer</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>corrupt ret addr and saved EBP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Code Injection</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>Return-to-libc</td>
<td></td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>ROP in libraries</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>ROP in application code</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
</tr>
</tbody>
</table>

* by combining PIC/PIE and ASLR
Put Them All Together

- Although none of them protects against everything, putting them all together protects against almost all attacks, even if there is a normal buffer overflow.
  - Do they really?
- However, buffer overflows bugs can have any forms, and some of these bugs are too-buggy to defend with these defense techniques.
- Attackers can be quite resourceful when exploiting a specific buffer overflow bug.
Blind ROP

- Something defeats them all (Bittau et al. 2014)
- Still needs a buffer overflow bug
- Only applicable to processes that restarts its threads after an error
  - Almost all modern server applications do this
- Find a buffer overflow
  - Remote fuzzy testing: send a large string to the server, see if it crashes. If yes, there is buffer overflow
  - This is actually the common way of finding buffer overflow besides reading source code
Blind ROP cont'd

- Defeat Stack canary and ASLR – Stack reading:
  - Take stack canary for example: write a random value to the last byte of canary, see if the thread crashes. If yes, try a new value to the last byte; if not, this value is correct, try next byte
  - Essentially, try different values until the thread does not crash, then a correct value is found
  - Can be generally applied to find randomized text segment of the program.
Blind ROP cont'd

- Remote fuzz testing to find ROP gadgets in program text segment:
  - Specific gadgets must be found
  - Write random return addresses to program's text segment
  - Program will jump to the random address to execute
  - Inspect program behavior to determine if a gadget is found
- Eventually enough gadgets are found, and shellcode can be launched
Blind ROP Limitations

- Only applicable to processes that restart its threads after an error
  - Almost all modern server applications do this

- No re-randomization after restart
  - Windows does re-randomization (but system libraries are not re-randomized)

- Moral of the story: the war between defend and attack will never stop
Performance vs Security

• There are also a lot of good protection techniques, many of them have the potential to provide full protection

• However, they incur too much overhead

• The trade-off between performance and security is a very important system engineering issue