Introduction to Cyber Security

Fall 2017 | Sherman Chow | CUHK IERG 4130

Chapter 2
Application Security
(Control Hijacking via Buffer Overflow)

Control Hijacking Attacks

- "Take over" the target machine (*e.g.*, a web server)
  - Taking over means executing arbitrary code
- by hijacking application control flow
- Example: Buffer overflow attacks
  - Morris worm, the 1st major exploit in 1988 Internet worm
  - Exploited a gets() call in "fingerd"
- Other Example: Format string vulnerabilities
  - What does %s mean in C? A format string to be replaced by an expression
  - Attacker supplies, e.g., %x, replacing %s, to read some memory content
  - *e.g.*, printf, fprintf, sprintf, vprintf, vfprintf, vsprintf, syslog, err, warn

Buffer Overflow

- Memory errors in C and C++ programs are among the oldest classes of software vulnerabilities.
- Single biggest software security threat
- The most common form of vulnerability till ’05 or so
- >25 years of independent, academic, & industry-related research
- Even if we consider only classic buffer overflows,
  it has been lodged in the top-3 most dangerous software errors for years
  - CWE/SANS Top-25 Most Dangerous Software Errors
- Buffer overflow vulnerabilities dominate in the area of remote network penetration vulnerabilities

Buffer Overflow in C/C++

- Stack is a memory region for the caller function to "communicate" with the callee function (*e.g.*, input of callee)
- BO: store more data in the buffer (stack / heap) than it can hold
- The next contiguous chunk of memory is overwritten
- C/C++ language is inherently **unsafe**
  - It allows programs to overflow buffers at will
  - No runtime checks that prevent writing pass the end of a buffer
  - strcpy(buf, "this string takes 27 bytes"); // buf only has 12 bytes

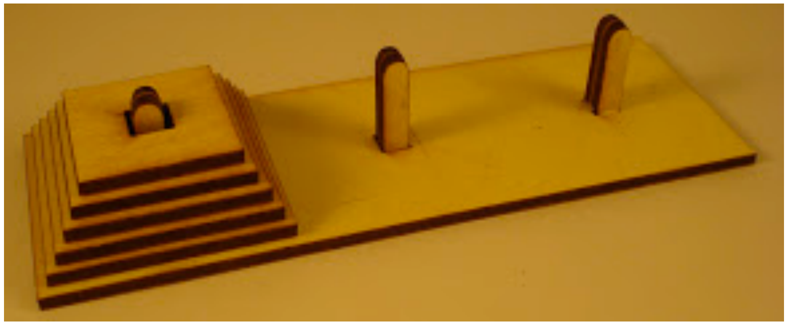
Buffer

- Typically, a program has 4 different areas of memory.
- Code area stores the compiled program.
- Global area stores the global variables.
- Heap, where dynamically allocated variables are allocated from
  - the kind of data when you call “malloc()” or “new”
- Stack, where parameters and local variables are allocated from

```
int func( ) {
        char buf[12];       // a buffer of 12 bytes
…
```

Stack

- Stack is a last-in-first-out (LIFO) data structure.
  - Push (put things in) and Pop (take things out)
- (Poor analogy!) Computer's memory is already there
  - We do not really add an empty box one after one
- Stack is implemented with a "marker" known as "stack pointer"
- Anything below the marker is considered "on the stack"
- Anything at the marker or above it
  - is not on the stack
  - may not erase the data there



Stack as a data structure for function call

- Pushed when calling a function and popped when returning
- Base (frame) pointer – points to a fixed location within a frame
- When a function is called, the following are pushed
  - function arguments, we need to pass the input to the function
  - the return address, we need to know where to go back
  - stack frame pointer
    - we need a reference of address to locate other variables
    - not to be confused w/ stack pointer
    - stack pointer will change, and hence it is not useful to be a reference
  - the local variables
  - (in that order)

Concrete Example

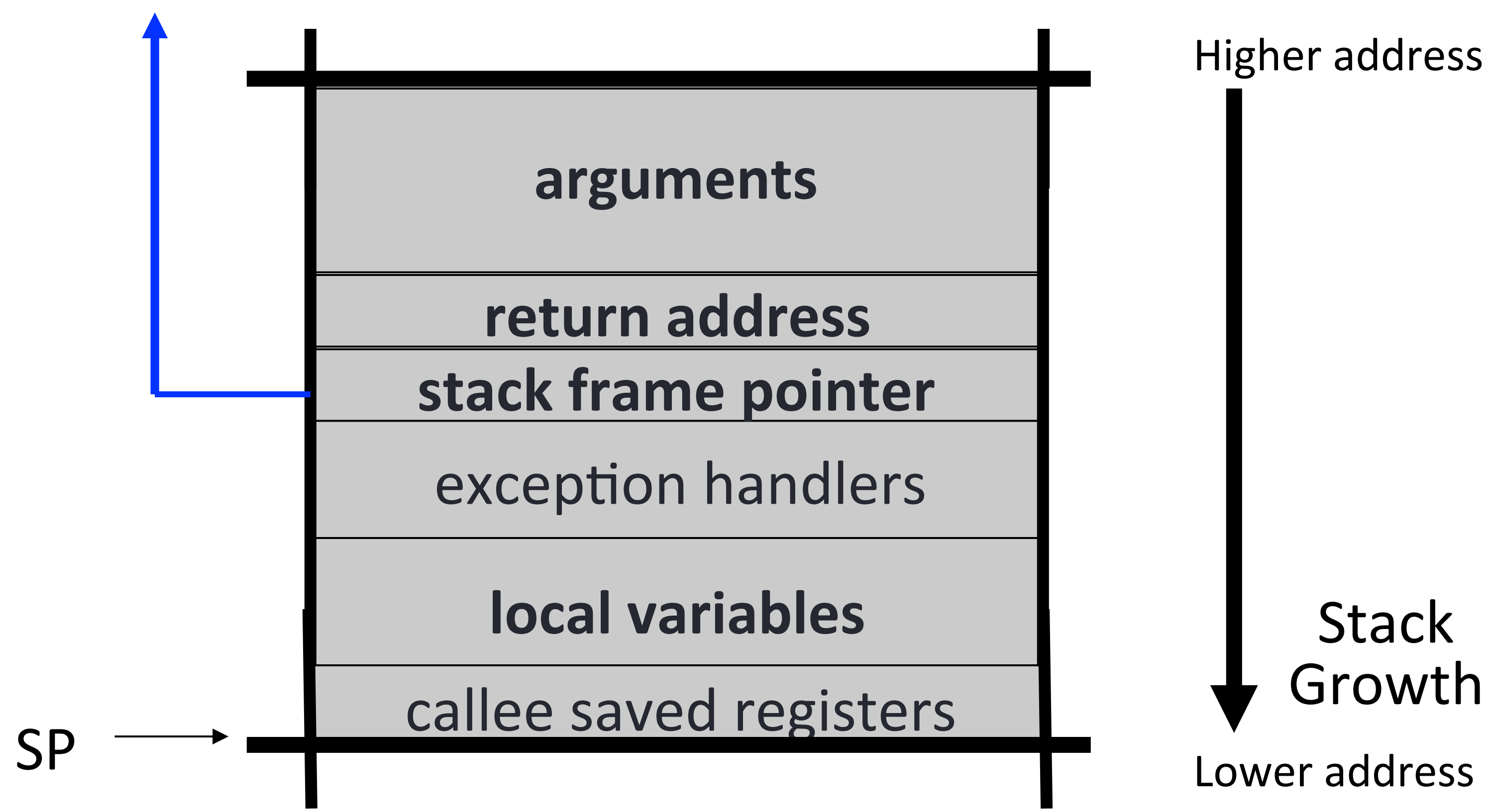
- Values of Arguments Input to the Function (str for f1( ))
- Address to Return to when the Function call is completed
- Memory space for Local (Function) Variables (buf[128])
- Way to Restore (clean-up) the Stack
  - looks the same before the call
- Starting address of Function call code
  - in the Code Area a.k.a. Text segment

```
Void f1(char *str){
   char buf[128] ;

   // codes for the function

}
```

Stack depicted



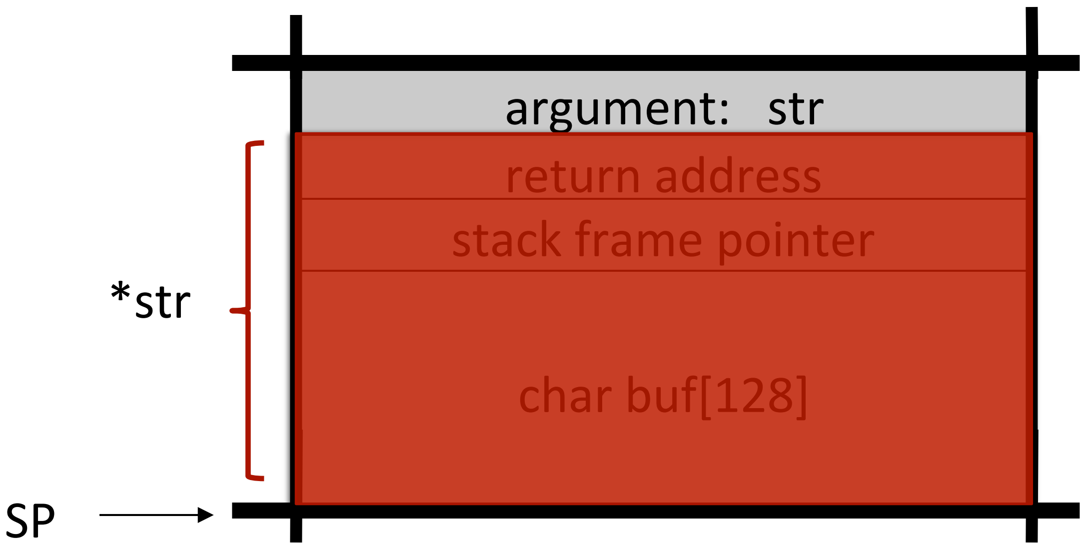
[these few slides are adopted from those of Stanford CS155]

Two Required Tasks to Realize The Attack

- Inject the attack code into a running process
  - typically a small sequence of instructions that spawns a shell
  - with shell (*e.g.*, bash, tcsh) then you can do many things
- Change the execution path of the running process to execute the attack code
  - *i.e.*, overwrite the return address so it points to the attack code
- Overflowing stack buffers can achieve both objectives simultaneously.

First Step of Buffer Overflow

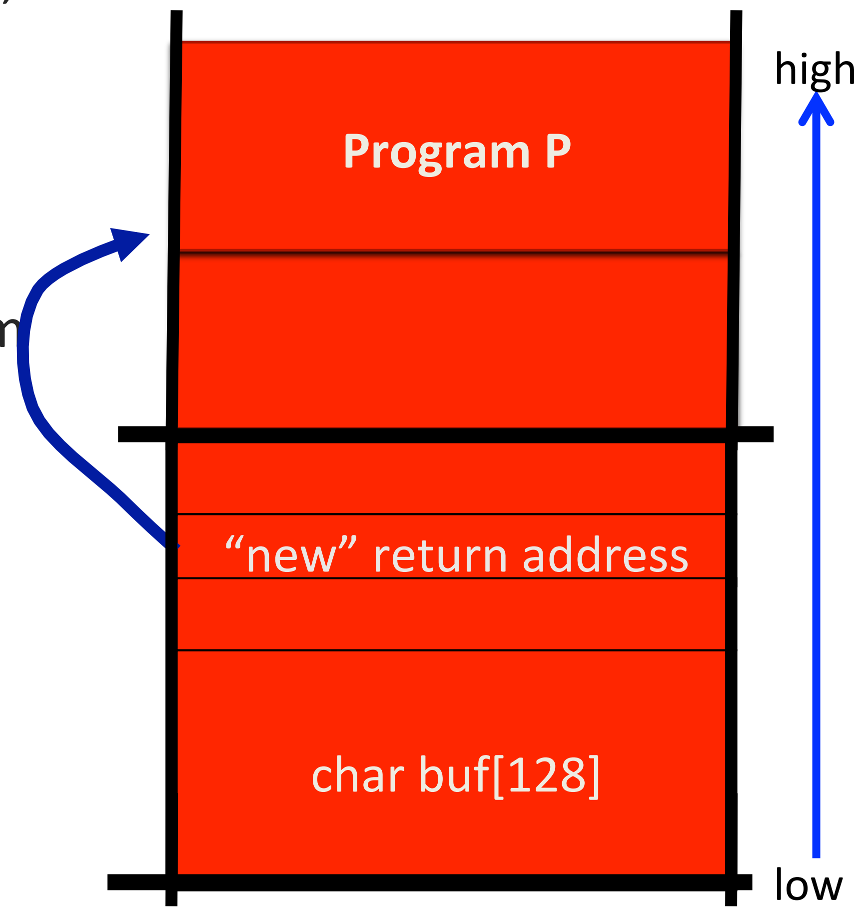
- What if we supply a long (>128) str such that it'll overflow buf?
- The problem occurs since there is no checking by strcpy()
- What's the use if an attacker can modify the return address?




```
Void f1(char *str){
   char buf[128] ;
   strcpy (buf, str);
   // other codes which
   // possibly "process" buf
}
```

Second Step of BO Attack

- Suppose *str is such that after strcpy, stack looks like:
- Program P: *e.g.*, exec("/bin/sh")
  - Note: attack code P runs *in stack*.
- Consider this is a web-server program
- When f1() exits, user gets the shell!
  - May inherit all permissions



Other Technical Issues

- How does the attacker guess precisely where is the start of P?
  - Insert many NOP (no operation) before P
  - More details covered in the tutorial (and the references)
- Other complications:
  - Program P should not contain the '\0' character. (Recall: it is a string)
  - Overflow should not crash program before f1() exits (Recall: seg. fault)
- Details vary slightly between CPUs and OSs:
  - Little endian vs. Big endian (x86 vs. Motorola)
  - Stack Frame structure (Unix vs. Windows)

Example of Unsafe libc Functions

- strcpy (char *dest, const char *src)
- strcat (char *dest, const char *src)
- gets (char *s)
- scanf (const char *format, …)
- "Safe" libc versions strncpy(), strncat() are misleading
  - *e.g.*, strncpy() may leave string unterminated.
- Windows C run time (CRT):
  - strcpy_s (*dest, DestSize, *src): ensures proper termination

Ways to find BO-vulnerable program

- Segmentation fault (core dumped)
- Run the program on local machine
- Issued malformed request (say, it is a web-server)
  - ending with some special string, like "$$$$$", if server crashes…
  - search core dump for "$$$$$" to find overflow locations
  - or automated tools (fuzzers)
    - https://www.owasp.org/index.php/Fuzzing
- Who bother local machine? Know yourself and your enemy…

Other BO Opportunities [**]

- We talked about Stack Overflow, how about Heap Overflow?
- Exception handlers (*e.g.*, Windows SEH attacks)
- Function pointers (*e.g.*, PHP 4.0.2, MS MediaPlayer Bitmaps)
- Longjmp buffers (*e.g.*, Perl 5.003)

Defenses

- Use *Safe* functions / rewrite in a type-safe language (*e.g.*, Java)
  - Difficult for existing (legacy) code
- Perform security-focused code review
- It’s about overflow! Let’s check the string’s length!
  - But… integer overflow (if you are summing up two string-lengths)
- It’s about overflow! Let’s check if “my stuff” is over-written
  - Stack canaries (a canary will warn about toxic gas in coal mine)

Canary Types

- Random canary
  - randomly choose a small integer
  - place it just before the return pointer
  - check if this value is overwritten
    - check here and there… performance degrades a little bit
  - If so, warn/exit program
    - Exiting program… potential exploit to launch a DoS attack?
- Terminator canary
  - String functions will not copy beyond terminator \0
  - Attacker cannot use string functions to corrupt stack.
  - Other examples: \n (new line), linefeed, EOF (end of file)

More Defenses

- Check whether a code is residing in an allow-to-be-executed segment before executing it
- Use other security checking-tools which will guard against array-boundary-overflow at run-time
- Stack randomization
  - *e.g.*, pad random bytes between return address and local buffers
  - difficult to predict the distance between them
  - custom attack for every copy of the randomized binary
- Some require compiler-support, but executable is compiled already

Ongoing Race ./. Attackers and Defenders

- Non-executable-Stack Features
  - From operating systems' support, *e.g.*, Solaris O.S.
  - From hardware support (cannot turn this "switch" off)
  - Protected region in memory: W^X (either writable or executable)
  - "only" place for shellcode (code for exploit) payload is non-protected region
- "Return to libc" attack can circumvent "Non-executable Stack"
  - libc is a shared library which provides the C runtime on Unix-style sys.
  - libc always links to the program and func. like system() is v. useful for attack
- Return-Oriented Programming: Exploits Without Code Injection
  - First presented at BlackHat USA Briefing '08, later in CCS '10
- Countermeasures: Address space layout randomization (ASLR) [**]

References

- Smashing the Stack for Fun and Profit
  - http://insecure.org/stf/smashstack.html
- Buffer Overflows: Attacks and Defenses for the vulnerability of the Decade
  - https://users.ece.cmu.edu/~adrian/630-f04/readings/cowan-vulnerability.pdf
  - Invited talk at SANS '00
- Basic Integer Overflows
  - http://phrack.org/issues/60/10.html#article
- Bypassing Browser Memory Protections
  - www.blackhat.com/presentations/bh-usa-08/Sotirov_Dowd/bh08-sotirov-dowd.pdf
- Heap Feng Shui in JavaScript
  - www.blackhat.com/presentations/bh-europe-07/Sotirov/Presentation/bh-eu-07-sotirov-apr19.pdf