CS526: Computer Security

Fall 2015

Topic 8
Software Security

Secure Software

- **"A program is secure"** – What does it mean?
- To understand program security one has to understand if the program behaves as its designer intended and as the user expected
- Software plays
    - a major role in providing security
    - as source of insecurity

Why Software Vulnerabilities Matter?

- When a process **reads input from attacker**, the process may be exploited if it contains vulnerabilities.
- When an attacker successfully exploits a vulnerability, he can
  - Crash programs: Compromises **availability**
  - Execute arbitrary code: Compromises **integrity**
  - Obtain sensitive information: Compromises **confidentiality**
- Software vulnerability enables the attacker to run with **privileges of other users**, violating desired **access control policy**

Attacks Exploiting Software Vulnerabilities

- Drive-by download (drive-by installation)
  - malicious web contents exploit vulnerabilities in browsers (or plugins) to download/install malware on victim system
- Email attachments in PDF, Word, etc.
- Network-facing daemon programs (such as http, ftp, mail servers, etc.) as entry points
- Privilege escalation
  - Attacker on a system exploits vulnerability in a root process and gains root privilege

Secure Code – Where do we stand today?

Critical crypto bug leaves Linux, hundreds of apps open to eavesdropping

This GnuTLS bug is worse than the big Apple "goto fail" bug patched last week.

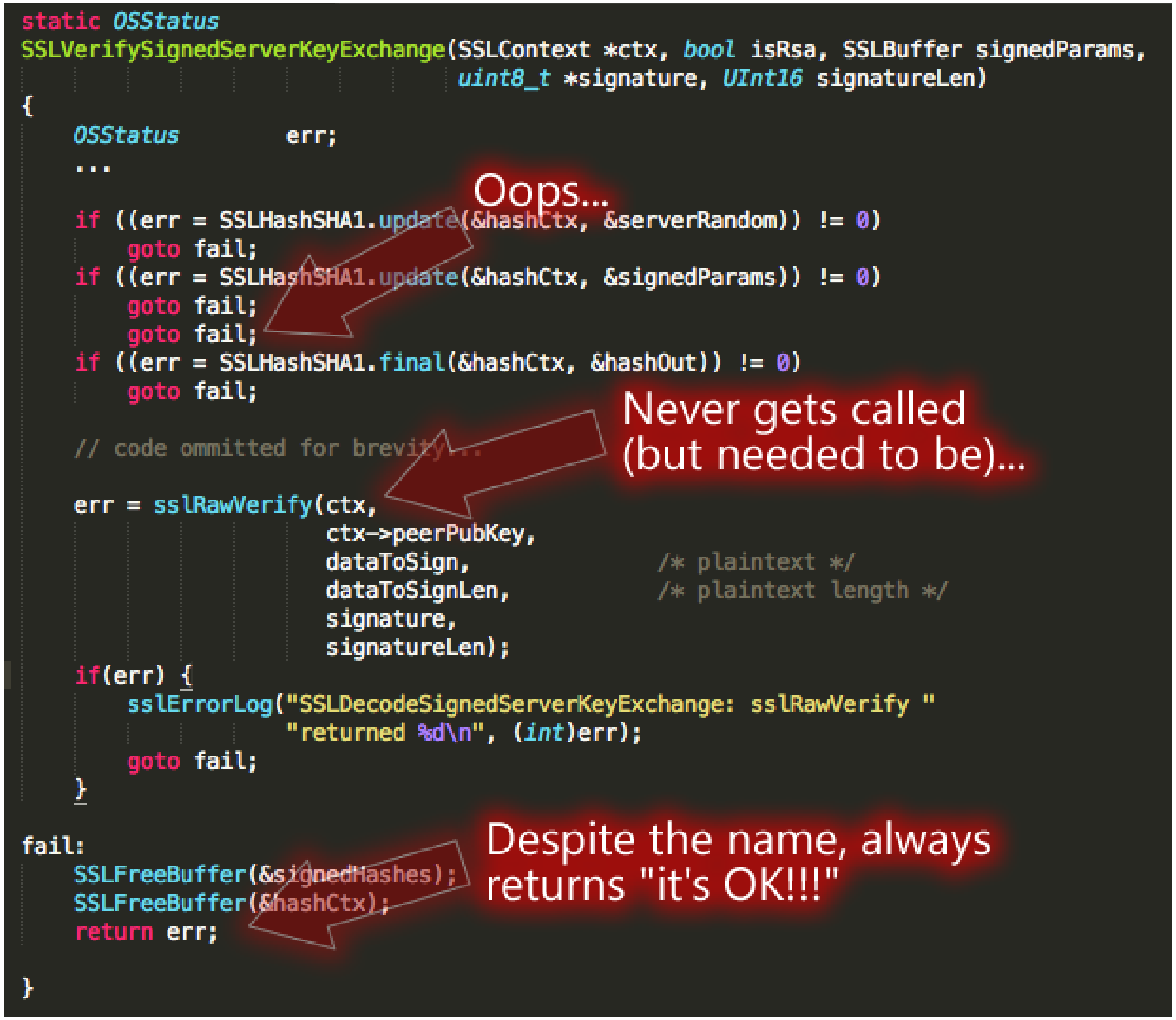
by **Dan Goodin** - Mar 4 2014, 1:56pm EST

Microsoft Security Advisory (2588513)

The Internet of Things Is Wildly Insecure — And Often Unpatchable

BY MICHAEL EISEN 01.06.14 | 6:30 AM | PERMALINK

A Real Example of Vulnerability



```
static OSStatus
SSLVerifySignedServerKeyExchange(SSLContext *ctx, bool isRsa, SSLBuffer signedParams,
                                 uint8_t *signature, UInt16 signatureLen)
{
    OSStatus        err;
    ...

    if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
        goto fail;
    if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
        goto fail;
        goto fail;
    if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
        goto fail;

    // code ommitted for brevity...

    err = sslRawVerify(ctx,
                       ctx->peerPubKey,
                       dataToSign,                /* plaintext */
                       dataToSignLen,             /* plaintext length */
                       signature,
                       signatureLen);
    if(err) {
        sslErrorLog("SSLDecodeSignedServerKeyExchange: sslRawVerify "
                    "returned %d\n", (int)err);
        goto fail;
    }

fail:
    SSLFreeBuffer(&signedHashes);
    SSLFreeBuffer(&hashCtx);
    return err;

}
```

Oops...

Never gets called
(but needed to be)...

Despite the name, always
returns "it's OK!!!"

** This picture is taken from http://zd.net/1mLOuXz

Common Software Vulnerabilities

- **Input validation**
- **Race conditions**
  - **Time-to-check-to-time-to-use (TOCTTOU)**
- **Buffer overflows**
- Format string problems
- Integer overflows
- Failing to handle errors
- Other exploitable logic errors

Input validation

Sources of Input that Need Validation

- Sources of input for local applications
  - Command line arguments
  - Environment variables
  - Configuration files, other files
  - Inter-Process Communication call arguments
  - Network packets

- Sources of input for web applications
  - Web form input
  - Scripting languages with string input

Environment variables

- Users can set the environment variables to anything
  - Using execve
  - Has some interesting consequences
- Examples:
  - PATH
  - LD_LIBRARY_PATH
  - IFS

Attack by Resetting PATH

- A setuid program has a system call: system(ls);
- The user sets his PATH to be . (current directory) and places a program ls in this directory
- The user can then execute arbitrary code as the setuid program

- **Solution**: Reset the PATH variable to be a standard form (i.e., "/bin:/usr/bin")

Attack by Resetting IFS

- Attacker can reset the IFS variable
  - IFS is the characters that the system considers as white space

- If not, the user may add “s” to the IFS
  - system(ls) becomes system(l)
  - Place a function l in the directory

- Moral: things are intricately related and inputs can have unexpected consequences

Attack by Resetting LD_LIBRARY_PATH

- Assume you have a setuid program that loads dynamic libraries
- UNIX searches the environment variable LD_LIBRARY_PATH for libraries
- A user can set LD_LIBRARY_PATH to /tmp/attack and places his own copy of the libraries here
- Most modern C runtime libraries have fixed this by not using the LD_LIBRARY_PATH variable when the EUID is not the same as the RUID or the EGID is not the same as the RGID

Command Line as Source of Input

```
void main(int argc, char** argv) {
    char buf[1024];
    sprintf(buf, “cat %s”,argv[1]);
    system (“buf”);
}
```

Intention: get a file name from input and then cat the file

- **What can go wrong?**
  - Attacker can add to the command by using, e.g., “**a; ls**”

Input Validation in Web Applications

- A remote example (PHP passthru)

```
echo 'Your usage log:<br />';
$username = $_GET['username'];
passthru(“cat /logs/usage/$username”);
```

- PHP **passthru(string)** executes command
- **What can go wrong?**
  - Attackers can put “;” to input to run desired commands, e.g., “`username=John;cat%20/etc/passwd`”

Directory Traversal Vulnerabilities in Web Applications

- A typical example of vulnerable application in php code is:

```
<?php
   $template = 'red.php';
   if ( isset( $_COOKIE['TEMPLATE'] ) )
   $template = $_COOKIE['TEMPLATE'];
   include ( "/home/users/phpguru/templates/" . $template );
?>
```

- Attacker sends

```
GET /vulnerable.php HTTP/1.0
Cookie: TEMPLATE=../../../../../../../../../etc/passwd
```

Checking input can be tricky:
Unicode vulnerabilities

- Some web servers check string input
  - Disallow sequences such as ../ or \
  - But may not check unicode %c0%af for '/'
- IIS Example, used by Nimda worm

http://victim.com/scripts/../../winnt/system32/cmd.exe?<some command

  - passes <some command> to cmd command
  - scripts directory of IIS has execute permissions
- Input checking would prevent that, but not this

http://victim.com/scripts/..%c0%af..%c0%afwinnt/system32/...

  - IIS first checks input, then expands unicode

Input Validation in Web Applications

- SQL injection
  - Caused by failure to validate/process inputs from web forms before using them to create SQL queries

- Cross Site Scripting
  - Caused by failure to validate/process inputs from web forms or URL before using them to create the web page

Takeaway: Input Validation

- **Malicious inputs** can become code, or change the logic to do things that are not intended
- Inputs interact with each other, sometimes in subtle ways
- Use **systematic approaches** to deal with input validation
  - Avoid checking for bad things (**blacklisting**)
  - Instead check for things that allowed (**whitelisting**)

Time-of-check-to-time-of-use

- **TOCTTOU**, pronounced "*TOCK too*“
- A class of software bug caused by changes in a system between the checking of a condition (such as authorization) and use of the results of the check.
  - When a process P requests to access resource X, the system checks whether P has right to access X; the usage of X happens later
  - When the usage occurs, perhaps P should not have access to X anymore.
  - The change may be because P changes or X changes.

An Example TOCTTOU

- In Unix, the following C code, when used in a setuid program, is a TOCTTOU bug:

```
if (access("file", W_OK) != 0)
            { exit(1); }

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

Attacker tries to execute the following line in another process when this process reaches exactly this time:
Symlink("/etc/passwd", "file")

- Here, *access* is intended to check whether the real user who executed the setuid program would normally be allowed to write the file (i.e., *access* checks the real userid rather than effective userid).

TOCTTOU

- Exploiting a TOCTTOU vulnerabilities requires precise timing of the victim process.
  - Can run the attack multiple times, hoping to get lucky
- Most general attack may require “single-stepping” the victim, i.e., can schedule the attacker process after each operation in the victim
  - Techniques exist to “single-step” victim
- Preventing TOCTTOU attacks is difficult

Buffer overflow

What is a Buffer Overflow?

- Buffer overflow occurs when a program or process tries to store more data in a buffer than the buffer can hold

- **Very dangerous** because the extra information may:
  - Affect user's data
  - Affect user's code
  - Affect system's data
  - Affect system's code

Why Does Buffer Overflow Happen?

- **No checks on bounds**
  - Programming languages give user too much control
  - Programming languages have unsafe functions
  - Users do not write safe code

- **C** and **C++**, are **more vulnerable** because they provide no built-in protection against accessing or overwriting data in any part of memory
  - Can’t know the lengths of buffers from a pointer
  - No guarantees strings are null terminated

Why Buffer Overflow Matters

- Overwrites
  - other buffers
  - variables
  - program flow data

- Results in
  - erratic program behavior
  - a memory access exception
  - program termination
  - incorrect results
  - **breach of system security**

History

- Used in 1988’s Morris Internet Worm
- Alphe One’s “**Smashing The Stack For Fun And Profit**” in Phrack Issue 49 in 1996 popularizes stack buffer overflows
- Still extremely common today

*The Internet Worm Program: An Analysis --- by Eugene H. Spafford (http://spaf.cerias.purdue.edu/tech-reps/823.pdf)

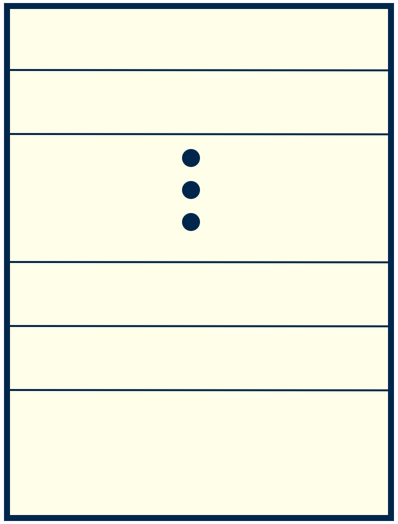
Types of Buffer Overflow Attacks

- Stack overflow
    - Shell code
    - Return-to-libc
        - Overflow sets ret-addr to address of libc function
    - Off-by-one
    - Overflow function pointers & longjmp buffers

- Heap overflow

Process Memory

- **A 32-bit process sees memory as an array of bytes that goes from address 0 to 2^{32}-1 (0 to 4GB-1)**

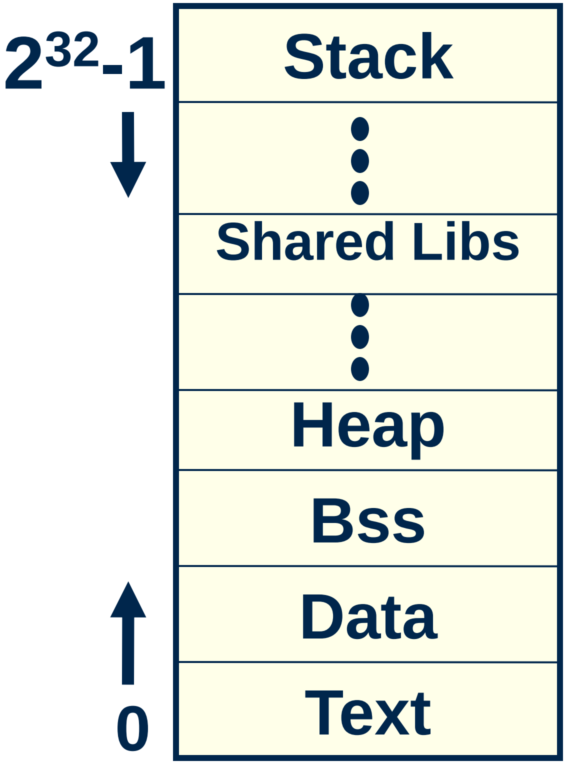


(4GB-1) 2^{32}-1

0

Memory Sections

The memory is organized into sections called "memory mappings"



Memory Sections

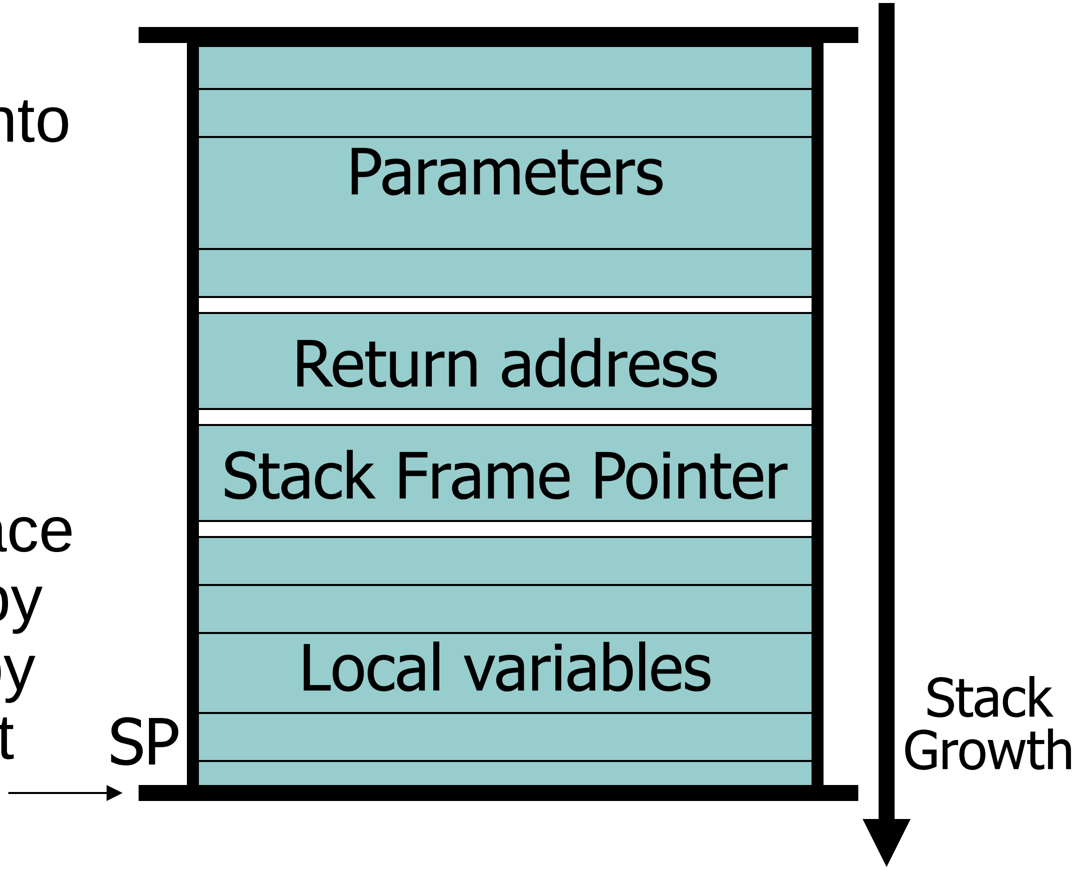
- **Each section has different permissions: read/write/execute or a combination of them.**
- **Text- Instructions that the program runs**
- **Data – Initialized global variables.**
- **Bss – Uninitialized global variables. They are initialized to zeroes.**
- **Heap – Memory returned when calling malloc/new. It grows upwards.**
- **Stack – It stores local variables and return addresses. It grows downwards.**

Background: C Program Execution

- PC (**program counter** or instruction pointer) points to next machine instruction to be executed
- Procedure call
    - Prepare parameters
    - Save state (SP (stack pointer) and PC) and allocate on stack local variables
    - Jumps to the beginning of procedure being called
- Procedure return
    - Recover state (SP and PC (this is return address)) from stack and adjust stack
    - Execution continues from return address

Background: Stack Frame

- Parameters for the procedure
- Save current PC onto stack (**return address**)
- Save current SP value onto stack
- Allocates stack space for local variables by decrementing SP by appropriate amount



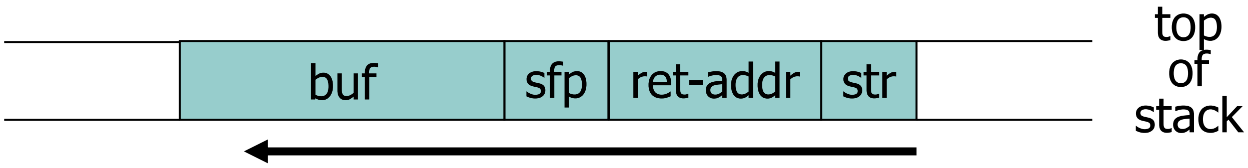
Example of a Stack-based Buffer Overflow

- Suppose a web server contains a function:

```
void my_func(char *str) {
  char buf[128];

  strcpy(buf, str);
  do-something(buf);
}
```

- When the function is invoked the stack looks like:



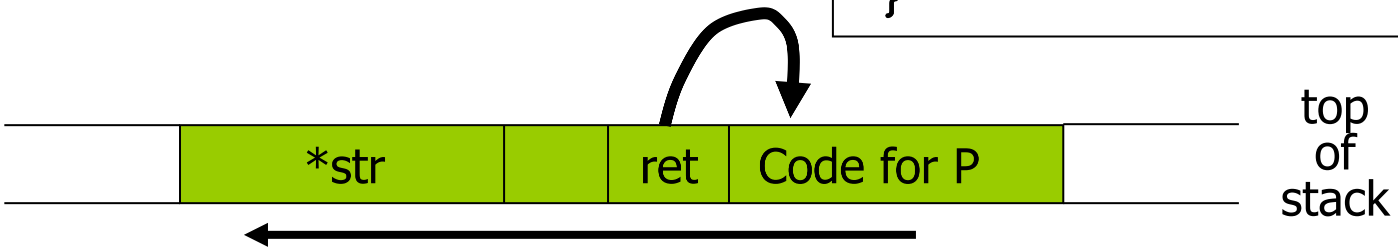
- What if `*str` is 136 bytes long? After `strcpy`:

Basic Stack Exploit

```
void my_func(char *str) {
   char buf[128];

   strcpy(buf, str);
   do-something(buf);
}
```

- Suppose ***str** is such that after **strcpy** stack looks like:



Program P: **`exec( "/bin/sh" )`**

- When **my_func()** exits, the user will be given a shell
- Note: attack code runs ***in stack***.
- To determine **ret** attacker guesses position when my_func() is called.

For more info, see **Smashing the Stack for Fun and Profit** by Aleph One

Carrying out this Attack Requires

- Determine the location of injected code position on stack when my_func() is called
  - So as to change **RET** on stack to point to it
  - Location of injected code is fixed relative to the location of the stack frame
- Program P should not contain the ‘\0’ character.
  - Easy to achieve
- Overflow should not crash program before my_func() exits

Some Unsafe C lib Functions

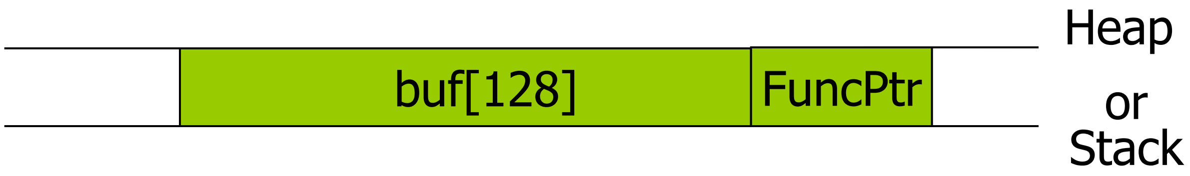
```
strcpy (char *dest,  const char *src)
strcat (char *dest, const char *src)
gets (char *s)
scanf ( const char *format, … )
printf (conts char *format, … )
```

⋮

Other Control Hijacking Opportunities

- Stack smashing attack (the basic stack attack)
  - Overwrite return address on the stack, by overflowing a local buffer variable.

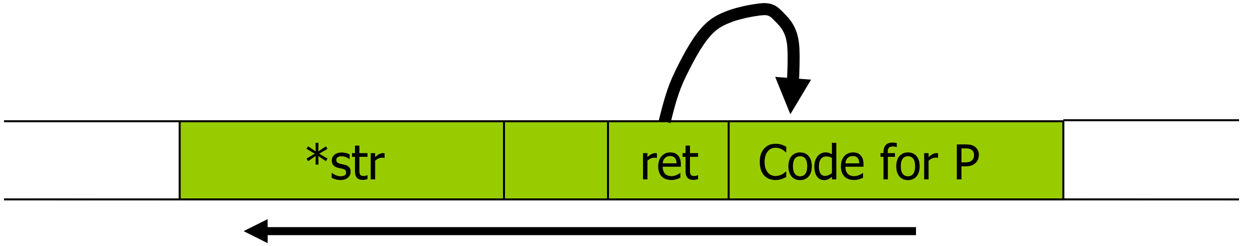
- Function pointers (used in attack on PHP 4.0.2)



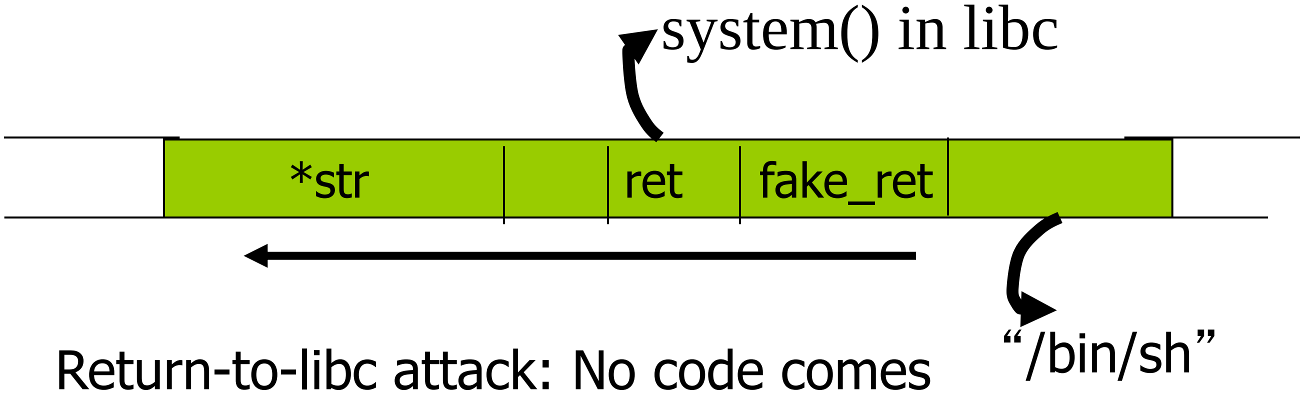
  - Overflowing buf will overwrite function pointer.

return-to-libc attack

- “Bypassing **non-executable-stack** during exploitation using return-to-libc” by c0ntex



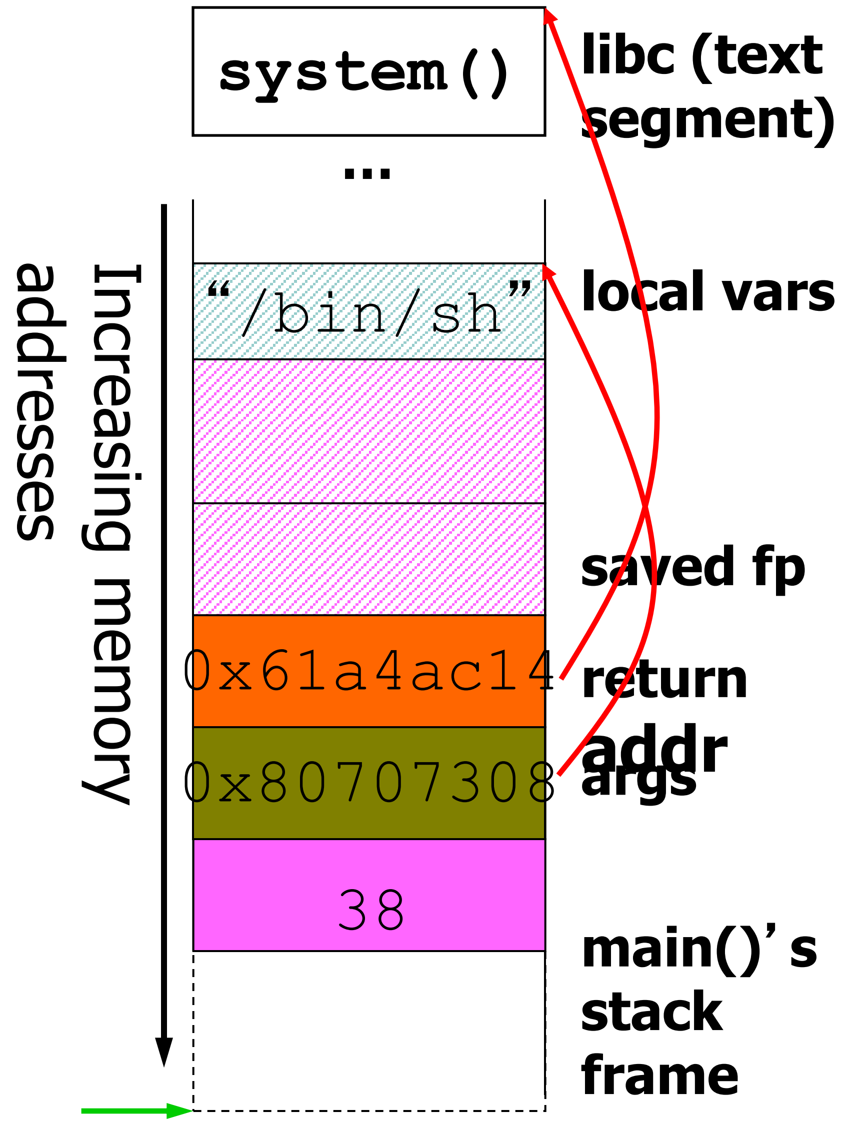
Shell code attack: Program P: **`exec( “/bin/sh” )`**



Return-to-libc attack: No code comes after ret (only the arg for the call)

Return-to-libc Attacks

- Instead of putting shellcode on stack, can put args there, overwrite return address with pointer to well known library function
  - e.g., **`system("/bin/sh");`**
- Return-to-libc attack



Slide from Brad Karp, UCL.

Heap-based Buffer Overflow Attacks

- Remember that heap represents data sections other than the stack
  - buffers that are dynamically allocated, e.g., by **malloc**
  - statically initialized variables (data section)
  - uninitialized buffers (bss section)

- Heap overflow may overwrite other data allocated on heap
- By exploiting the behavior of memory management routines, attacker may overwrite an arbitrary memory location with a small amount of data

Prevention mechanisms

Preventing Buffer Overflow Attacks

- Use type safe languages (e.g., Java)
- Use safe library functions (e.g., strncpy)
- Static source code analysis
- Non-executable stack
- Run time checking (e.g., StackGaurd)
- Address space layout randomization (ASLR)
- Detecting deviation of program behavior

Static Source Code Analysis

- Statically check source code to detect buffer overflows
- **Automate** the code review process
- Several tools exist
- **Expensive** (exponential)
- Typically done for short programs of critical importance
- Find lots of bugs, but not all

Bugs to Detect in Source Code Analysis

- Some examples

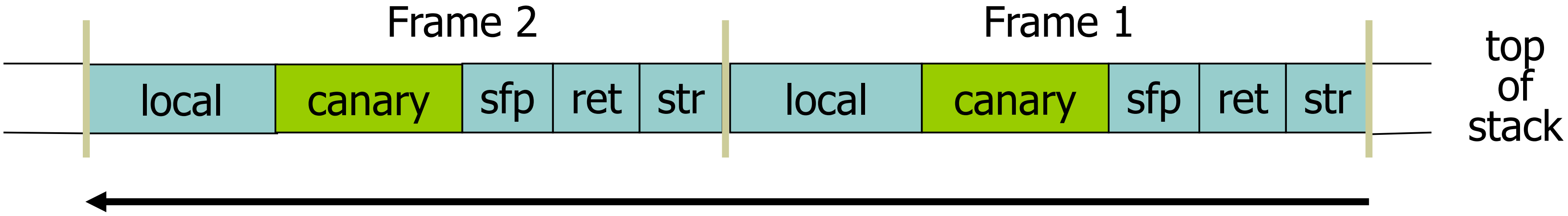
  - Crash Causing Defects
  - Null pointer dereference
  - Use after free
  - Double free
  - Array indexing errors
  - Mismatched array new/delete
  - Potential stack overrun
  - Potential heap overrun
  - Return pointers to local variables
  - Logically inconsistent code
  - Uninitialized variables
  - Invalid use of negative values
  - Passing large parameters by value
  - Underallocations of dynamic data
  - Memory leaks
  - File handle leaks
  - Network resource leaks
  - Unused values
  - Unhandled return codes
  - Use of invalid iterators

Non-Executable Stack

- Basic stack exploit can be prevented by **hardware support** to mark stack segment as non-executable
    - Support in Windows since XP SP2. Code patches exist for Linux, Solaris.
- Problems:
    - Does not defend against all attacks (see “return-to-libc”)
    - Does not block more general overflow exploits
        - Overflow on heap; overflow func pointer

Run Time Checking: StackGuard

- StackGuard checks for stack integrity at run time
  - E.g., embed “**canaries**” in stack frames and verify their integrity prior to function return.



Canary Types

- **Random canary**
  - Choose random string at program startup
  - Insert canary string into every stack frame
  - Verify canary before returning from function
  - To corrupt random canary, attacker must learn current random string
- **Terminator canary**
  - Canary = 0, newline, linefeed, EOF
  - String functions will not copy beyond terminator.
  - Hence, attacker cannot use string functions to corrupt stack.
  - Weakness: Adversary knows canary
- Canaries **do not offer** full protection

Other Run Time Checking

- Validate sufficient space (LibSafe)
  - E.g., intercept calls to strcpy (dest, src) and check that:
    `|frame-pointer - dest| > strlen(src)`
  - If so do strcpy, else terminate application.
- Copying to a safe location (StackShield)
  - E.g., at function prologue, copy return address to a safe location, and upon return check that return address still equals the saved copy

Randomization: Motivations

- Buffer overflow and return-to-libc exploits **need to know the (virtual) address** to which pass control
  - Address of attack code in the buffer
  - Address of a standard kernel library routine

- Same address is used on many machines
  - Slammer infected 75,000 MS-SQL servers using same code on every machine
- Idea: introduce **artificial diversity**
  - Make stack addresses, addresses of library routines, etc. unpredictable and different from machine to machine

Address Space Layout Randomization

- Arranging the positions of key data areas randomly in a process' address space.
  - e.g., the base of the executable and position of libraries (libc), heap, and stack,
  - Effects: for return to libc, needs to know address of the key functions.
  - Attacks:
    - Repetitively guess randomized address
    - Use non-ASLR modules
- Supported on Windows Vista, Linux (and other UNIX variants)

Takeaway

- Software vulnerabilities may have severe implications
- Mostly result from improper input validation and buffer overflow
- Avoid using functions that don’t check boundaries

Acknowledgement

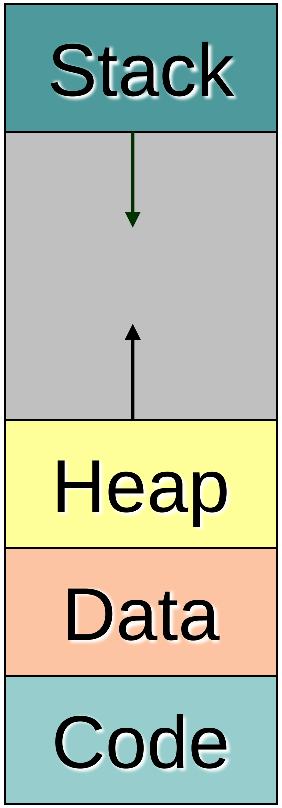
Slides from Ninghui Li, Endadul Haque, and Cristina Nita-Rotaru

Thank you

Background: Programs and Memory

- The operating system creates a process by assigning memory and other resources
- **Code**: the program instructions to be executed
- **Data**: initialized variables including global and static variables, un-initialized variables
- **Heap**: dynamic memory for variables that are created (e.g., with *malloc)* and disposed of with *free*
- **Stack**: keeps track of the point to which each active subroutine should return control when it finishes executing; stores variables that are local to functions

Virtual Memory



Code Fragment for Printing Stack Frame (from prstack.c)

```
int fac(int a, int p) {
 char f[8] = "        ";
 int b = 0;
  // print stack frame
 gets(f);      // buffer may overflow
 if (a == 1) {  b = 1; }
 else {  b = a * fac(a-1,p); }
   // print stack frame again }
 return b;
}
```

```
int main(int argc, char*argv[]) {
  int n;
  int r;
 if (argc == 2) {
   n = atoi(argv[1]);
   r = fac(n, 0);
 } else if (argc == 3) {
   n = atoi(argv[2]);
   r = fac(n, 1);
 }
 return 0;
}
```

Code Fragment for Printing Stack Frame (from prstack.c)

```
int fac(int a, int p) {
        char f[8] = "       "; int b = 0;
 printf("Address %p: argument  int     p: 0x%.8x\n", &p, p);
 printf("Address %p: argument  int     a: 0x%.8x\n", &a, a);
 printf("Address %p: return address     : 0x%.8x\n", &a-1, *(&a-1));
 printf("Address %p: saved stack frame p: 0x%.8x\n", &a-2, *(&a-2));
 printf("Address %p: local var f[4-7]   : 0x%.8x\n", (char *)(&f)+4,

       *((int *)(&f[4])));
 printf("Address %p: local var f[0-3]   : 0x%.8x\n", &f, *((int *)f));
 printf("Address %p: local var int     b: 0x%.8x\n", &b, b);
 printf("Address %p:   gap              : 0x%.8x\n", &b-1, *(&b-1));
…
}
```

Printed Stack Frame

```
•Entering function call fac(a=2), code at 0x080484a5
•Address 0xff98942c: argument  int     p: 0x00000001
•Address 0xff989428: argument  int     a: 0x00000002
•Address 0xff989424: return address     : 0x0804860e
•Address 0xff989420: saved stack frame p: 0xff989440
•Address 0xff98941c: local var f[4-7]   : 0x00202020
•Address 0xff989418: local var f[0-3]   : 0x20202020
•Address 0xff989414: local var int     b: 0x00000000
•Address 0xff989410:   gap              : 0x00000000

•Entering function call fac(a=1), code at 0x080484a5
•Address 0xff98940c: argument  int     p: 0x00000001
•Address 0xff989408: argument  int     a: 0x00000001
•Address 0xff989404: return address     : 0x0804860e
•Address 0xff989400: saved stack frame p: 0xff989420
•Address 0xff9893fc: local var f[4-7]   : 0x00202020
•Address 0xff9893f8: local var f[0-3]   : 0x20202020
•Address 0xff9893f4: local var int     b: 0x00000000
•Address 0xff9893f0:   gap              : 0x00000000
```

Stack Frame with Overflowed Buffer

```
•Entering function call fac(a=1), code at 0x080484a5
•Address 0xffd5724c: argument  int     p: 0x00000001
•Address 0xffd57248: argument  int     a: 0x00000001
•Address 0xffd57244: return address     : 0x0804860e
•Address 0xffd57240: saved stack frame p: 0xffd57260
•Address 0xffd5723c: local var f[4-7]   : 0x00202020
•Address 0xffd57238: local var f[0-3]   : 0x20202020
•Address 0xffd57234: local var int     b: 0x00000000
•Address 0xffd57230:   gap              : 0x00000000
•123456789012345
```

Input 15 bytes

```
•Leaving function call fac(a=1), code at 0x80484a5
•Address 0xffd5724c: argument  int     p: 0x00000001
•Address 0xffd57248: argument  int     a: 0x00000001
•Address 0xffd57244: return address     : 0x00353433
•Address 0xffd57240: saved stack frame p: 0x32313039
•Address 0xffd5723c: local var f[4-7]   : 0x38373635
•Address 0xffd57238: local var f[0-3]   : 0x34333231
•Address 0xffd57234: local var int     b: 0x00000001
•Address 0xffd57230:   gap              : 0x00000001
•Segmentation fault (core dumped)
```

Overflowing f to overwrite saved sfp and

What does a function do?

- **`fac`**
- `0x080484a5 <+0>:      push   %ebp` save stack frame pointer (fp)
- `0x080484a6 <+1>:      mov    %esp,%ebp` set current stack fp
- `0x080484a8 <+3>:      sub    $0x18,%esp` allocate space for local var
- `0x080484ab <+6>:      movl   $0x20202020,-0x8(%ebp)` initialize f[0-3]
- `0x080484b2 <+13>:     movl   $0x202020,-0x4(%ebp)` initialize f[4-7]
- `0x080484b9 <+20>:     movl   $0x0,-0xc(%ebp)` initialize b
- `0x080484c0 <+27>:     mov    0xc(%ebp),%eax` load value of p to eax
- `0x080484c3 <+30>:     test   %eax,%eax` check if eax is 0
- `0x080484c5 <+32>:     je     0x80485e8 <fac+323>` if so, skip printing frame
- `....`
- `0x080485e8 <+323>:    mov    0x8(%ebp),%eax` load value of a to eax
- `0x080485eb <+326>:    cmp    $0x1,%eax` check if a==1
- `0x080485ee <+329>:    jne    0x80485f9 <fac+340>` if not, call fac