SWE 781
Secure Software Design and Programming

Buffer Overflows

Lecture 4

Ron Ritchey, Ph.D.
Chief Scientist
703/377.6704
Ritchey_ronald@bah.com

Copyright Ronald W. Ritchey 2008, All Rights Reserved

Readings for this lecture

- Chest & West
  - Chapters 6 & 7
- Wheeler
  - Chapter 6
- Additional readings
  - Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade, Crispin Cowan, et al.
  - Smashing The Stack For Fun And Profit, Aleph One
  - Beyond Stack Smashing: Recent Advances in Exploiting Buffer Overruns. Pincus and Baker.

Copyright Ronald W. Ritchey 2008, All Rights Reserved

Today's Agenda

- Buffer Overflow Sources
- Buffer Overflow Attack Mechanics
- Possible system-level solutions

Copyright Ronald W. Ritchey 2008, All Rights Reserved

Buffer overflows

- Extremely common bug.
  - First major exploit: 1988 Internet Worm. fingerd.
- 15 years later: ≈ 50% of all CERT advisories:
  - 1998: 9 out of 13
  - 2001: 14 out of 37
  - 2003: 13 out of 28
- Often leads to total compromise of host.
- Developing buffer overflow attacks:
  - Locate buffer overflow within an application.
  - Design an exploit.

Copyright Ronald W. Ritchey 2008, All Rights Reserved

Examples

- (In)famous: Morris worm (1988)
  - gets() in fingerd
- Code Red (2001)
  - MS IIS .ida vulnerability
- Blaster (2003)
  - MS DCOM RPC vulnerability
- Mplayer URL heap allocation (2004)

```
% mplayer http://`perl –e ‘print
  “\””x1024;’`
```

What is a Buffer Overflow?

- Intent
  - Arbitrary code execution
    - Spawn a remote shell or infect with worm/virus
  - Denial of service
- Steps
  - Inject attack code into buffer
  - Redirect control flow to attack code
  - Execute attack code

IATAC

Attack Possibilities

- Targets
  - Stack, heap, static area
  - Parameter modification (non-pointer data)
    - E.g., change parameters for existing call to `exec()`
- Injected code vs. existing code
- Absolute vs. relative address dependencies
- Related Attacks
  - Integer overflows, double-frees
  - Format-string attacks

Buffer Overflows

- Extremely common programming flaw.
  - Causes difficult to debug problems
  - Also a leading cause of security vulnerabilities
- Caused when a program attempts to store more data in a buffer than it can hold.

```
char buf[4];
strcpy(buf, “abcd”);
```

Copyright Ronald W. Ritchey 2008, All Rights Reserved

Exploitable Buffer Overflows

- When the source of the data is controlled by the user/attacker

```
#include <stdio.h>
int main(int argc, char * argv[]) {
     char name[26];
     printf("Please type your name: \n");
     gets(name);
     printf("Your name is %s\n", name);
}
```

OK

```
$./name
Please type your name: Ron Ritchey
```

Overflow

```
$./name
Please type your name: AAAAAAAAAAAAAAAAAAAAA
   AAAAABAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

Beginning of overflow

Copyright Ronald W. Ritchey 2008, All Rights Reserved

C string storage

- Much of the problem with buffer overflows comes from the way that C represents strings
  - As an array of chars
  - Terminated by a NIL
- Many system calls count on the NIL value to properly terminate
- Allocating space for strings can be misleading as you must explicitly leave space for the NIL

```
#define MAXNAME 7
char str[MAXNAME] // Not long enough
strcpy(str, “Ritchey”); // “Ritchey” is 8 chars long
```

Copyright Ronald W. Ritchey 2008, All Rights Reserved

Dangerous library routines

- ‘C/C++’ are filled with routines that do not perform bounds checking
  - strcpy, strcat
    - Count on the NIL character to terminate
    - Will happily overwrite memory until a NIL is read from source
  - sprintf,etc
    - Must ensure destination buffer large enough to hold string that results from all of the input variables
  - gets, scanf, etc
    - Do not limit input to fixed size
  - Format string driven functions
    - printf,sprintf, etc.
    - Allow attackers to write arbitrary values into memory if they can influence content of format string

IATAC

Copyright Ronald W. Ritchey 2008, All Rights Reserved

sprintf

- Very difficult to construct patterns of a fixed length.
- Destination must be large enough to hold the largest possible result
- Symantecs of width/precision vary depending upon the type of variable

```
char buf[BUFFER_SIZE];
sprintf(buf, "%*s",  sizeof(buf)-1, "long-string");
     /* WRONG */
sprintf(buf, "%.*s", sizeof(buf)-1, "long-string");
     /* RIGHT */
```

Copyright Ronald W. Ritchey 2008, All Rights Reserved

Alternative functions that bound operations

- Many replacement functions exist which allow you to specify a maximum length
- strncpy(char *dst, char *src, size_t len)
  - Copies up to len bytes from src to dst
  - If src length >= to len, dst will NOT be NIL terminated
  - If src length < len, remainder of dst will be NIL filled
- strncat(char *dst, char *src, size_t len)
  - Appends up to len chars to end of dst
  - Like strncpy does not NIL terminate if length of src >= len
  - Be careful! len refers to the space remaining in dst

Copyright Ronald W. Ritchey 2008, All Rights Reserved

A better strncpy: strlcpy

- Available at ftp://ftp.openbsd.org/pub/OpenBSD/src/lib/libc/string/strlcpy.3.
- size_t strlcpy (char *dst, const char *src, size_t size);
  - Copies up to size-1 characters from the NUL-terminated string src to dst, NIL-terminating the result
  - dst is guaranteed to be nil terminated if size > 0
  - Size refers to total size of dst, not number of bytes from src
- size_t strlcat(char *dst, const char *src, size_t size)
  - Concatenates src to dst using same semantics as strlcpy

Copyright Ronald W. Ritchey 2008, All Rights Reserved

Bounding is not a panacea

- It is still possible to introduce buffer overflow errors when using bounded functions
  - Use a bad value for bound
    - Source length instead of destination length
    - Lack of room for NIL termination
  - Miscalculation of space available for concatenation (e.g. strncat)
    - Providing length of buffer instead of remaining space in buffer
  - Concatenated value changes semantics of use

```
void badfunc(char *s) {
  char buf[10];
  strncpy(buf, s, strlen(s));
}
```

Copyright Ronald W. Ritchey 2008, All Rights Reserved

Concatenation bounds must be based on calculation of remaining space available

- Even the bounded concatenation routines (e.g. strncat, strlcat) can easily overflow buffers
  - When given unterminated input
    - Routines search for first NIL in input to begin concatenation operation. If no NIL is provided, routines will seek past end of buffer until a NIL is reached in memory. This can cause very difficult to diagnosis failures
  - When bound value calculation is wrong
    - Bound value set to total size of variable instead of remaining size
    - Remaining size value calculation flawed

```
void badfunc() {
  char buf[10];
  char *s = "1234567890";
  strncpy(buf, s, sizeof(buf));
  strncat(buf, ";", sizeof(buf));
}

void betterfunc() {
  char buf[10];
  char *s = "1234567890";
  strncpy(buf, s, sizeof(buf));
  buf[sizeof(buf)-1] = '\0';
  strncat(buf, ";",
    10 - strlen(buf));
}
```

What's still wrong with this?

Copyright Ronald W. Ritchey 2008, All Rights Reserved

String truncation vulnerabilities

- Just limiting the length of the input may not be enough to prevent vulnerability. E.g.

```
fgets(line, 128, stdin);
// Check format
strncpy(buf, line, 12);
if (strncmp(“.mil”, line+strnlen(line,128)-4 , 4)) {
  // Allow access
}
```

- Input ABCDE123.milabcdefg will be accepted
- Always perform format checks just before use

Copyright Ronald W. Ritchey 2008, All Rights Reserved

Always make sure to terminate your strings

- Anytime there is a potential of truncation, make sure to terminate properly
  - Writing NIL to last possible value often a good safety method

IP->
```
char buf[20];
strncpy(buf, "Hello World",
  sizeof(buf));
buf[sizeof(buf)-1] = '\0';
```

H	e	l	l	o		W	o	r	l	d	\0								\0

```
strncpy(buf, "Hello World
  War Three", sizeof(buf));
buf[sizeof(buf)-1] = '\0';
```

H	e	l	l	o		W	o	r	l	d		W	a	r		T	h	r	e \0

IATAC

Copyright Ronald W. Ritchey 2008, All Rights Reserved

Format string vulnerabilities

- Format strings specify a set of formatting rules to be applied to create a string based upon a set of input variables

```
testfunc(char *varname, int varvalue) {
  printf("%s value is %2d", varname, varvalue);
}
```

- Never allow the user to control the format string

```
badfunc(char *s) {
      printf(s);
}
```

  - May allow attacker to read arbitrary data locations in memory
  - With use of %n directive may be able to write to memory
    - %n writes the number of characters processed so far to the address specified in the parameter list

      ```
      printf("ABC%n",number);
      ```

    - Overwriting any location of memory possible if attacker can control the value of n and the location of memory that n will be written to
    - May allow attacker to gain control of instruction pointer
      - E.g. overwrite ret value on stack, overwrite commonly called function pointer, etc.

IATAC

Copyright Ronald W. Ritchey 2008, All Rights Reserved

Watch out for multi-byte character formats

- To support foreign character sets, multi-byte character formats have been created
  - Unicode, UTF-8, UTF-16, ISO-8859-1
- Bytes per character vary based upon standard
  - Fixed Width – ISO-8859-1, UTF-32
  - Variable Width – UTF-8, UTF-16
- When using multi-byte functions, must ensure that correct type is used for size limitations
  - Bytes vs. Characters – Will not be the same for variable with formats

IATAC

Copyright Ronald W. Ritchey 2008, All Rights Reserved

Preventing Buffer Overflows

- Strategies
  - Detect and remove vulnerabilities (best)
  - Prevent code injection
  - Detect code injection
  - Prevent code execution
- Stages of intervention
  - Analyzing and compiling code
  - Linking objects into executable
  - Loading executable into memory
  - Running executable

Preventing Buffer Overflows

- Type safe languages (Java, ML)
  - Legacy code?
- Splint - Check array bounds and pointers
- Non-executable stack
- Stackguard – put canary before RA
- Libsafe – replace vulnerable library functions
- RAD – check RA against copy
- Analyze call trace for abnormality
- PointGuard – encrypt pointers
- Binary diversity – change code to slow worm propagation
- PAX – binary layout randomization by kernel
- Randomize system call numbers

IATAC

Today's Agenda

- Buffer Overflow Sources
- Buffer Overflow Attack Mechanics
- Possible system-level solutions

IATAC

Copyright Ronald W. Ritchey 2008, All Rights Reserved

What is needed

- Understanding C functions and the stack.
- Some familiarity with machine code.
- Know how systems calls are made.
- The exec() system call.
  - A way to run a new program in Unix
  - Does not create a new process, but changes a current process to a new program
  - What system call is needed to create a new process?
- Attacker needs to know which CPU and OS are running on the target machine.
  - Our examples are for x86 running Linux.
  - Details vary slightly between CPU’s and OS:
    - Stack growth direction.
    - big endian vs. little endian.

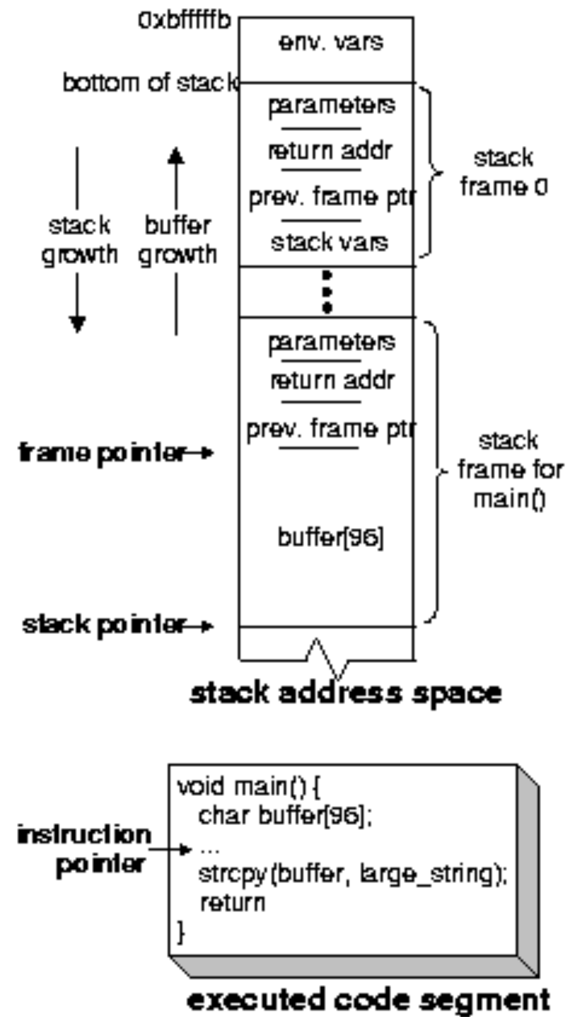
Copyright Ronald W. Ritchey 2008, All Rights Reserved

Steps to Smashing the Stack

- Inject machine code of exploit into heap or stack
- Cause running program to jump to this code
- Most common place to overflow is stack
  - Large amount of potential buffers allocated in local functions
  - Overwriting these buffers can also overwrite the return pointer
  - Careful attacker can overwrite the return pointer with the mem location of the exploit code
  - When the function RETs the program jumps to the start of the attack code

Copyright Ronald W. Ritchey 2008, All Rights Reserved

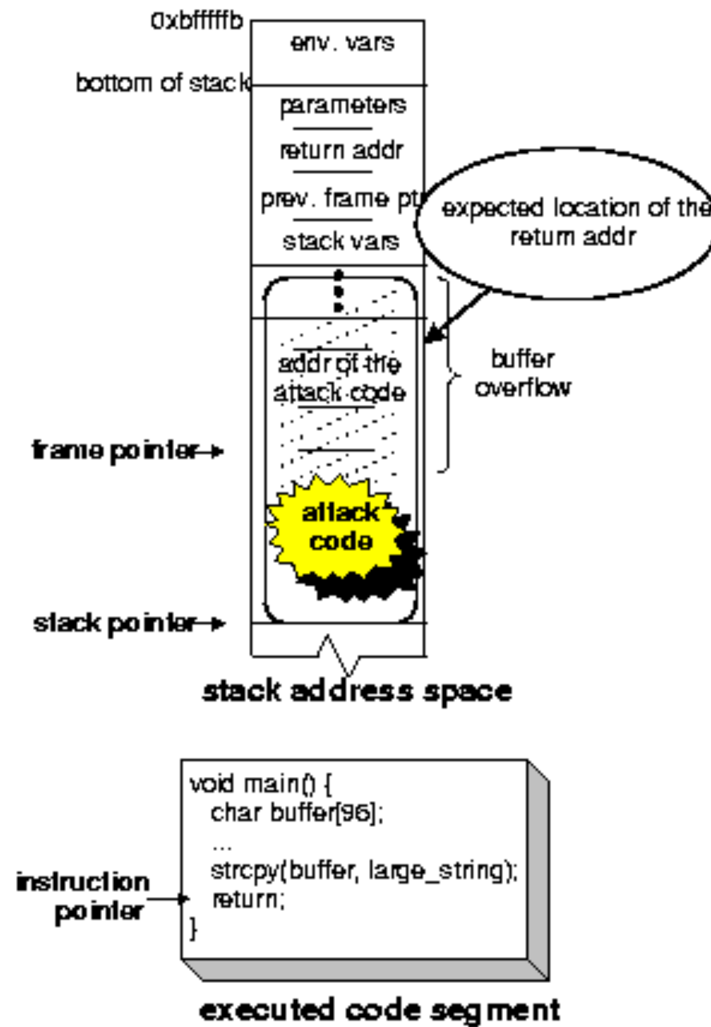
Stack Example: Before Attack



From Baratloo, et al., Transparent Run-Time Defense Against Stack Smashing Attacks

Copyright Ronald W. Ritchey 2008, All Rights Reserved

Stack Example: During Exploit



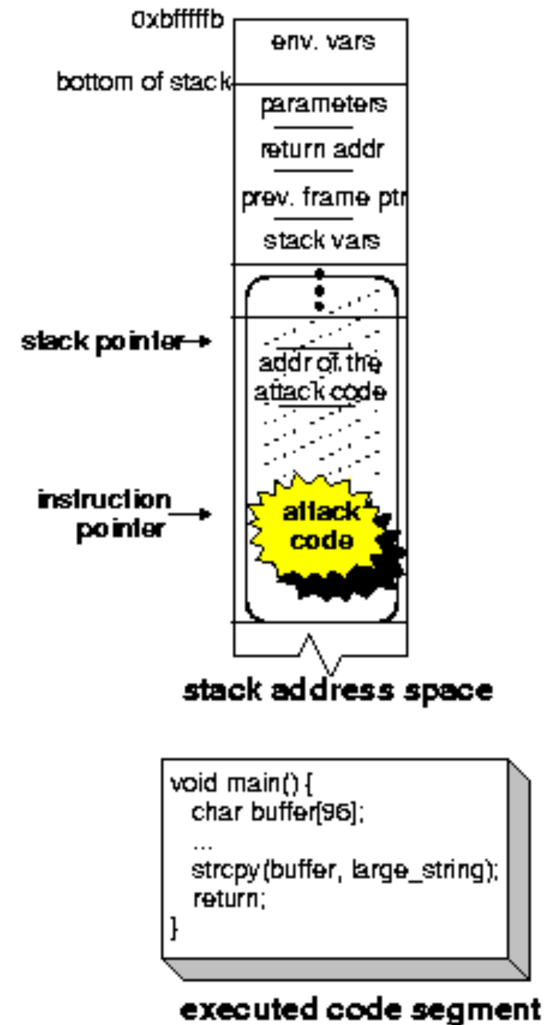
From Baratloo, et al., Transparent Run-Time Defense Against Stack Smashing Attacks

Copyright Ronald W. Ritchey 2008, All Rights Reserved

Stack Example: After attack

Attacker's Screen

```
#
#id
uid=0(root) gid=0(root) groups=0(
root),1(bin),2(daemon),3(sys),4 (
adm),6(disk),10(wheel)
```



From Baratloo, et al., Transparent Run-Time Defense Against Stack Smashing Attacks

Copyright Ronald W. Ritchey 2008, All Rights Reserved

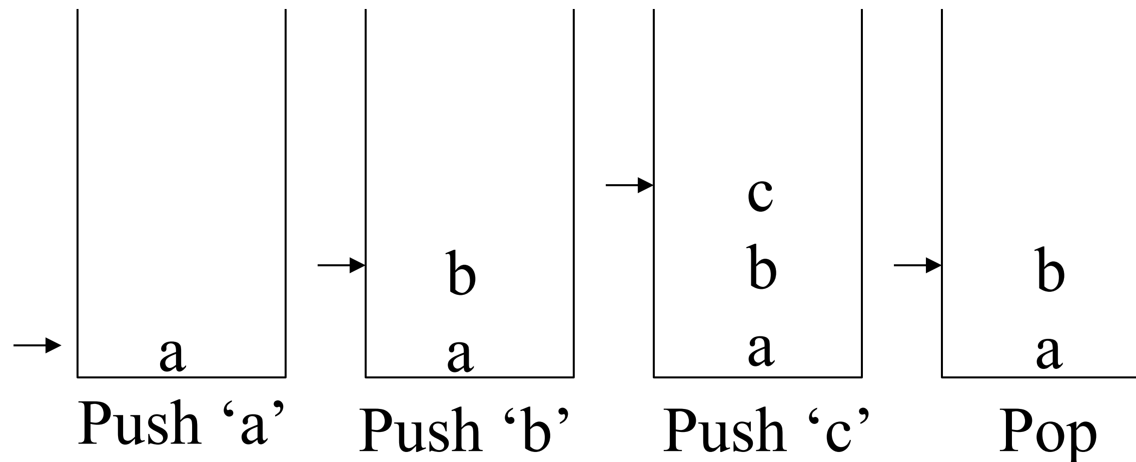
Buffer Overflows and the Stack

- To truly understand how a buffer overflow attack works you must understand the role the stack plays in a 3rd generation language function call
- Stacks are an essential part of computer science
- First-in/Last-Out storage
- Their use for holding onto information that needs to be retrieved FIFO make them a very convenient way of recording function variables.

Copyright Ronald W. Ritchey 2008, All Rights Reserved

Stacks

- A stack is a common data structure
  - Supports two main functions (Push, and Pop)
  - Push - Place data on stack
  - Pop - Retrieve data from stack



Copyright Ronald W. Ritchey 2008, All Rights Reserved

The Stack

- Many modern programming languages (include C/C++) use a stacks to help implement functions
- Functions
  - Like Gotos (jumps), alter the flow of execution
  - Unlike gotos allow the program to return control to the caller after a function is completed
- Stacks are used to store important details needed to allow control to return to the calling process

Copyright Ronald W. Ritchey 2008, All Rights Reserved

Why stacks?

- To allow functions to call functions
- If functions could only be one level deep, then a fixed data structure could be used to store the return information
- Since functions can call functions, it is important that all of the return information for each function call be saved
- Since depth of functions is not defined at compile time, it is important that the amount of memory that needs to be reserved for function variables is dynamically allocated

Copyright Ronald W. Ritchey 2008, All Rights Reserved

Important Registers

- EIP - Instruction Pointer
  - Points to location in memory that the CPU should execute next
- ESP - Stack Pointer
  - Points to current “top” of stack
- EBP - Frame Pointer
  - Used to efficiently reference local variables

IATAC

Copyright Ronald W. Ritchey 2008, All Rights Reserved

Function Call Walkthrough

- When a caller transfers execution to a function the following steps are taken
  - Arguments to function are pushed onto stack in reverse order
  - Address of the next instruction in the calling function is pushed on the stack
- The called function on start-up (prologue) must
  - Push current value of EBP onto the stack
  - Set EBP to current ESP value
  - Allocate space for local variables by moving the stack point enough to leave space for them

Copyright Ronald W. Ritchey 2008, All Rights Reserved

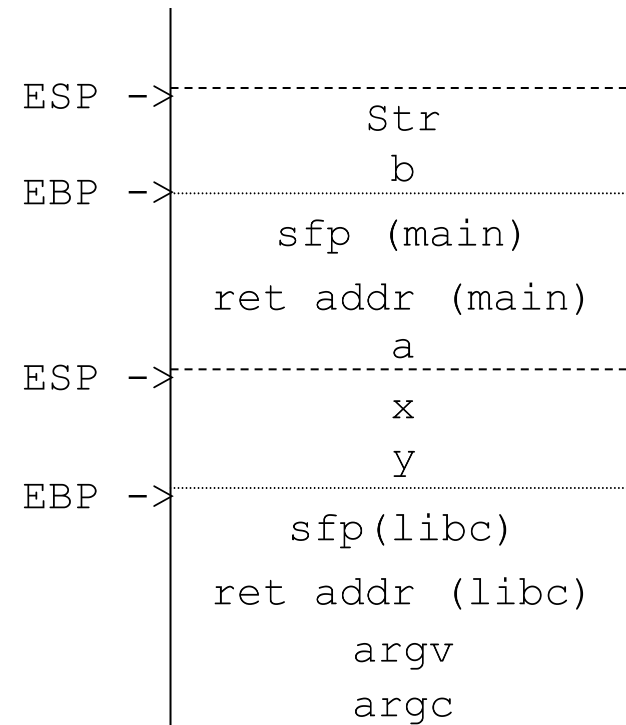
Function Return Walkthrough

- When the return occurs
  - Return value of function is saved in accumulator
  - ESP = EBP
  - Pop EBP (to restore Frame Buffer)
  - RET (EIP = Top of stack)

Copyright Ronald W. Ritchey 2008, All Rights Reserved

Stack Operation

```
 #include <stdio.h>
 int main(int argc, char *argv[])
 {
     int x;
     int y;
IP→  func(x);
 }

 int func(int a) {
    char str[10];
IP→ int b=2;

    strcpy(str, "Add A to B");
    printf("%s", str);
    return b + a;
 }
```



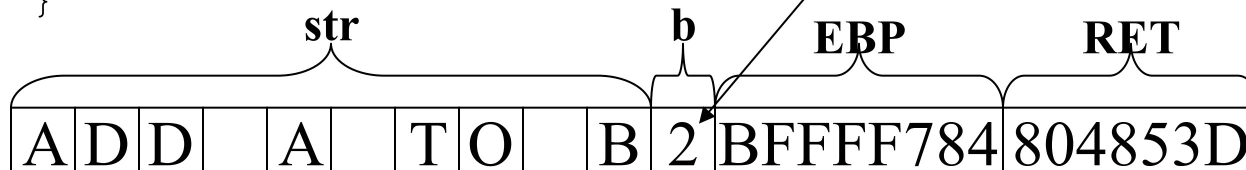
Copyright Ronald W. Ritchey 2008, All Rights Reserved

Overflowing the Stack

- Storing too much data in a variable causes the variable to overflow
- The extra data does not disappear! It is written to whatever is adjacent to the variable that has been overwritten.

```
int func(int a) {
  char str[10];
  int b=2;

  strcpy(str, "Add A to B");
  printf("%s", str);
  return b + a;
}
```

Is this value right?



Copyright Ronald W. Ritchey 2008, All Rights Reserved

Steps to Smashing the Stack

- Inject machine code of exploit into heap or stack
- Cause running program to jump to this code
- Most common place to overflow is stack
  - Large amount of potential buffers allocated in local functions
  - Overwriting these buffers can also overwrite the return pointer
  - Careful attacker can overwrite the return pointer with the mem location of the exploit code
  - When the function RETs the program jumps to the start of the attack code

IATAC

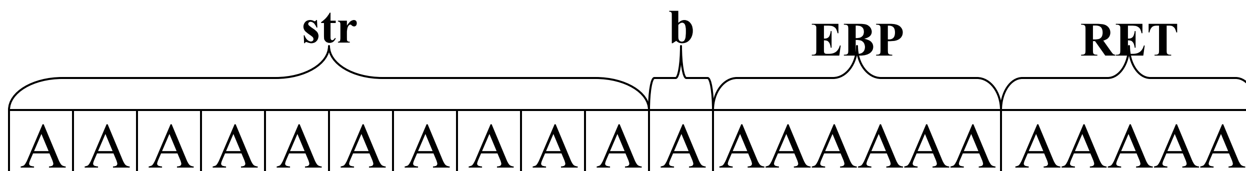
Copyright Ronald W. Ritchey 2008, All Rights Reserved

Overflow Past the EBP

- Will normally cause the program to crash as the RET value will normally point to a region of memory outside the program

```
int func(int a) {
  char str[10];
  int b=2;

  gets(str);
  printf("Type a string: ");
  printf("%s", str);
  return b + a;
}
```

```
$./testcode
Type a string: AAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAA
Segmentation fault (core dumped)
```



Copyright Ronald W. Ritchey 2008, All Rights Reserved

Manipulation of the RET value

- A careful attacker can overwrite the RET value with a valid location to return to
  - Program will go to this new location when function ends and will not (always) core dump

```
#include <stdlib.h>
void function(int a, int b, int c) {
        char buffer1[5];
        char buffer2[20];
        int *ret;
        ret = buffer1 + 12;
        (*ret) += 8;
}

void main() {
        int x;

        x = 0;
        function(1,2,3);
        x = 1;
        printf("%d\n",x);
}
```

```
$./test
0
```

Copyright Ronald W. Ritchey 2008, All Rights Reserved

How can rewriting RET be used?

- Denial of Service
- Jumping past authentication code!
- Accessing privileged system calls
- Gaining a shell prompt
  - By placing exploit code into a buffer
  - Rewriting RET to jump into the buffer
- Gaining a shell prompt
  - Finding a usable argument in memory
    - "/bin/sh"
  - Calling existing library routines to spawn a shell
    - execve

IATAC

Copyright Ronald W. Ritchey 2008, All Rights Reserved

A simple shellcode example

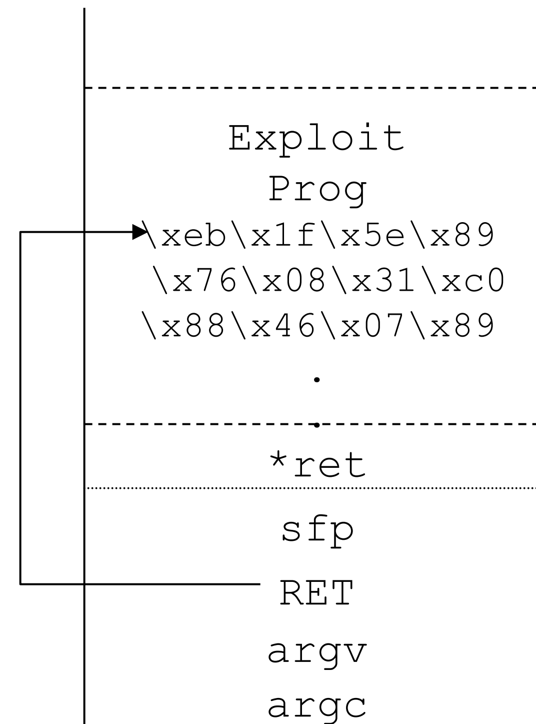
```
char shellcode[] =        "\xeb\x1f\x5e
\x89\x76\x08\x31\xc0\x88\x46\x07\x89”
“\x46\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c”
“\xcd\x80\x31\xdb\x89\xd8\x40\xcd\x80\xe8\xdc\xff”
“\xff\xff/bin/sh";

void main() {
    int *ret;
    ret = (int *) &ret + 2;
    (*ret) = (int)shellcode;
}
```

```
$ ls -l testsc
-rwsr-sr-x 1 root root 11450 Jun 10 10:07 testsc2*
$ ./testsc
# id
uid=0(root)
```

Copyright Ronald W. Ritchey 2008, All Rights Reserved

Stack diagram of exploit

- In this example the buffer was created by
  - Creating a buffer with the exploit program
  - Declaring a variable point ret.
  - Moving ret to the location in memory of the RET ptr
  - Overwriting RET with the start of the exploit code
    - `(*ret) = (int)shellcode;`
- In a real buffer overflow vulnerability, the attack would need a way to fill up the buffer from one of the program inputs



IATAC

Copyright Ronald W. Ritchey 2008, All Rights Reserved

Today's Agenda

- Buffer Overflow Sources
- Buffer Overflow Attack Mechanics
- Possible system-level solutions

Copyright Ronald W. Ritchey 2008, All Rights Reserved

Libsafe

- A replace library for some of the most common library functions that cause buffer overflows
- http://www.avayalabs.com/project/libsafe/index.html
- Protects return address by limiting stack access to the local stack

IATAC

Copyright Ronald W. Ritchey 2008, All Rights Reserved

Libsafe stack size check

- Libsafe determines at run-time the size of the stack by examining the current stack and frame pointers.
- If one of its wrapped functions attempts to write data to the stack that would overwrite the return address or any of the parameters it is denied.

IATAC

Copyright Ronald W. Ritchey 2008, All Rights Reserved

0xbfffffb

bottom of stack

env. vars
parameters
return addr
prev. frame ptr
stack vars
⋮
return addr
prev. frame ptr
attack code

frame pointer→

stack pointer→

limit buffer overflow to this region

stack address space

text region

```
void main() {
  char buffer[96];
  ...
  strcpy(buffer, large_string);
  return;
}
```

libsafe library

```
char *strcpy(char *dest, const char *src) {
  // compute length of input string
  // compute upper bound of destination's buffer size
  // bounds check
  // call libc's memcpy()
  // return
}
```

libc library

```
char * memcpy(void *dest, const void *src, size_t n) {
  ...
}

char *strcpy(char dest, const char *src) {
  ...
}
```

1 2 3 4

From Baratloo, et al., Transparent Run-Time Defense Against Stack Smashing Attacks

IATAC

Copyright Ronald W. Ritchey 2008, All Rights Reserved

Libsafe is not a perfect solution

- Implemented as a dynamic-link library.
  - Allows protection of previously compiled programs
  - Local attacker may be able to change to load order i.e. LD_PRELOAD to disable libsafe
- Only protects a limited number of library calls
- Only protects the return address on the stack. Heap overflows are still possible.
- As an application developer, you may not be able to rely on its presence.
- Can be confused by some compiler optimizations.

IATAC

Copyright Ronald W. Ritchey 2008, All Rights Reserved

Other solutions

- Turn off stack execution
  - Limited value as attackers may be able to easily find the calls they want already in the compiled program.
- Use a compiler that adds bounds-checking code
  - StackGuard (http://immunix.org/stackguard.html)
    - Adds “canary” value in front of return address
    - If canary overwritten, this return is not performed
- Use routines that manage Strings for you!
- Use languages that support dynamic memory management
  - Java, Perl, Python

Copyright Ronald W. Ritchey 2008, All Rights Reserved

Next Thursday's Class

Error Handling

Copyright Ronald W. Ritchey 2008, All Rights Reserved

Questions?

Copyright Ronald W. Ritchey 2008, All Rights Reserved