CIS 449/549: Software Security

Anys Bacha

Slides from U. Shankar, M. Hicks, K. Du, D. Boneh, N. Zeldovich, A. Rahmati

Memory Exploits

Memory Exploits

- We discussed stack smashing
  - Overflowing the buffer allocated onto the stack
  - Described by Aleph One's paper on *Smashing the Stack for Fun and Profit*

- Unlike buffer overflows, heap overflows involves overflowing a buffer allocated on the heap by malloc

Heap Overflow

- Overflowing a buffer allocated by malloc on the heap can result in:
  - Modify nearby data
  - Modify a secret key to a known value
  - Modify state to bypass authentication
  - Modify interpreted strings used in commands (e.g. SQL injection)
  - Modify a function pointer

Heap Overflow

- Overflow into the C++ object vtable
  - C++ objects (that contain virtual functions) are represented using a vtable, which contains pointers to the object's methods
- Overwrite heap metadata
  - Hidden header is located just before the pointer returned by malloc
  - If p = malloc(), then modifying *(p-1) corrupts the heap header
- Heap read overflow
  - Read data adjacent or nearby heap buffer
  - Leak secret info (e.g. Heartbleed)
  - Format string vulnerability

Heap Overflow Example

```
typedef struct _vulnerable_struct {
  char buff[MAX_LEN];
  int (*cmp)(char*,char*);
 } vulnerable;

int foo(vulnerable* s, char* one, char* two){
  strcpy( s->buff, one );
  strcat( s->buff, two );
  return s->cmp( s->buff, "file://foobar" );
}
```

Heap Overflow Example

```
typedef struct _vulnerable_struct {
  char buff[MAX_LEN];
  int (*cmp)(char*,char*);
 } vulnerable;

int foo(vulnerable* s, char* one, char* two){
  strcpy( s->buff, one );
  strcat( s->buff, two );
  return s->cmp( s->buff, "file://foobar" );
}
```

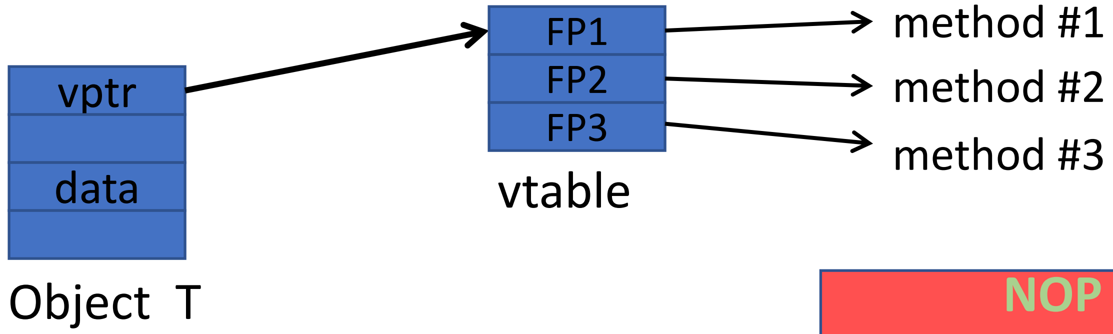
Unsafe functions $\longrightarrow$

s->cmp could be overwritten

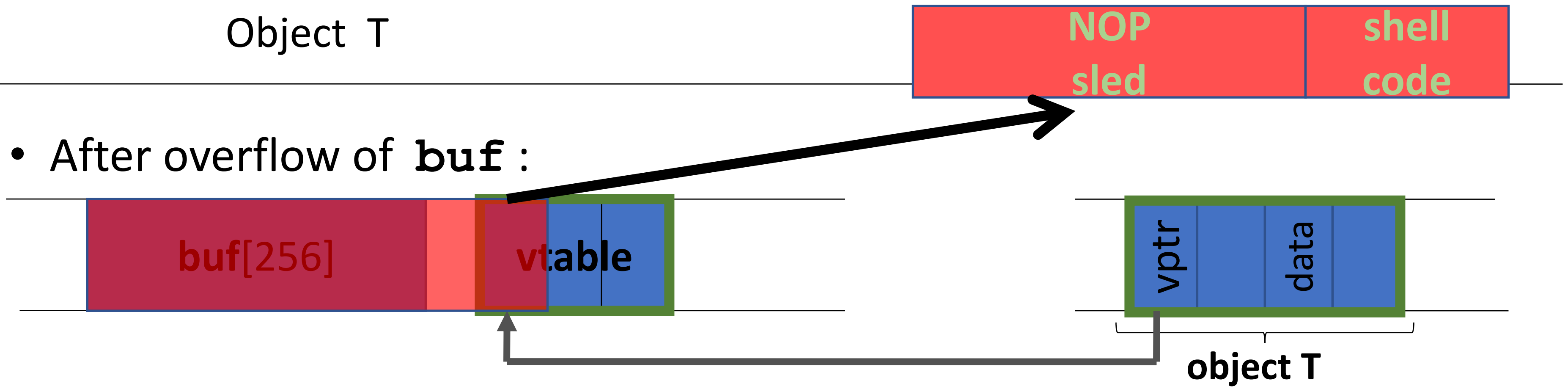
Must have strlen(one) + strlen(two) < MAX_LEN

Heap Overflow Example

- Compiler generated function pointers (e.g. C++ code)



- After overflow of **`buf`** :



Formatted I/O

- See notes

What’s Wrong with this Code?

```
#define BUF_SIZE 16
char buf[BUF_SIZE];
void vulnerable() {
 int len = read_int_from_network();
 char *p =  read_string_from_network();
 if(len > BUF_SIZE) {
      printf(“Too large\n”);
      return;
  }

  memcpy(buf, p, len);
}
```

What’s Wrong with this Code?

```
#define BUF_SIZE 16
char buf[BUF_SIZE];
void vulnerable() {
 int len = read_int_from_network();
 char *p =  read_string_from_network();
 if(len > BUF_SIZE) {
      printf(“Too large\n”);
      return;
  }
  memcpy(buf, p, len);
}
```

Negative $\longrightarrow$ (int len)

```
void *memcpy(void *dest, const void *src, size_t n);          typedef unsigned int size_t;
```

What’s Wrong with this Code?

```
               #define BUF_SIZE 16
               char buf[BUF_SIZE];
               void vulnerable() {
Negative  →     int len = read_int_from_network();
                char *p =  read_string_from_network();
OK        →     if(len > BUF_SIZE) {
                      printf(“Too large\n”);
                      return;              Implicit cast to unsigned
                 }                         ↙
                 memcpy(buf, p, len);
               }
```

void *memcpy(void *dest, const void *src, size_t n);

typedef unsigned int size_t;

Integer Overflows

Integer Overflows

Problem: what happens when int exceeds max value?

int m; (32 bits) **short s; (16 bits)** **char c; (8 bits)**

c = 0x80 + 0x80 = 128 + 128 ⇒ c = 0

s = 0xff80 + 0x80 ⇒ s = 0

m = 0xffffff80 + 0x80 ⇒ m = 0

Can this be exploited?

Integer Overflows

```
void  func( char *buf1, *buf2,    unsigned int len1, len2) {
     char temp[256];
     if  (len1 + len2 > 256)  {return;} // length check
     memcpy(temp, buf1, len1);        // cat buffers
     memcpy(temp+len1, buf2, len2);
     do-something(temp);              // do stuff
}
```

What if **len1 = 0x80,** **len2 = 0xffffff80** ?

$\Rightarrow$ len1+len2 = 0

Second memcpy() will overflow stack !!

What’s Wrong with this Code?

```
void vulnerable()
{
    size_t len;
    char *buf;
    len = read_int_from_network();
    buf = malloc(len + 5);
    read(fd, buf, len);
    ...
}
```

What’s Wrong with this Code?

```
void vulnerable()
{
    size_t len;
    char *buf;
    len = read_int_from_network();
    buf = malloc(len + 5);
    read(fd, buf, len);
    ...
}
```

Too big $\longrightarrow$ `len = read_int_from_network();`

`buf = malloc(len + 5);` $\longleftarrow$ **Wrap around**

You have to know the semantics of your programming language to avoid these errors

Defenses

Ways to Prevent Hijacking Attacks

- Fix bugs:
  - Audit software
    - Automated tools: Coverity, Valgrind, Prefast/Prefix.
  - Rewrite software in a safer language (e.g. Java)
    - Difficult for existing (legacy) code …
- Platform defenses: prevent attack code execution
- Add runtime code to detect overflows exploits
  - Halt process when overflow exploit detected
  - StackGuard

Challenges

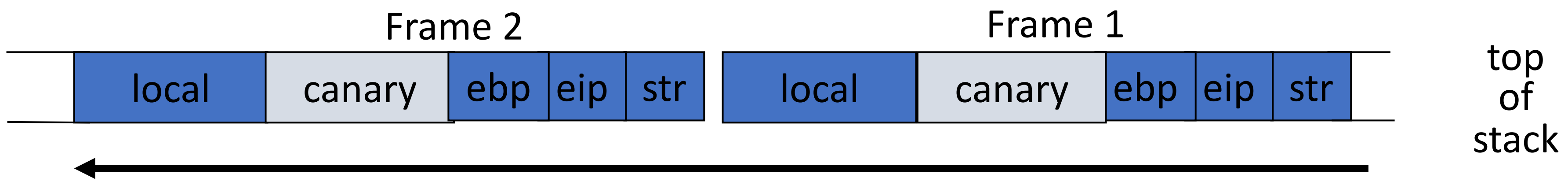
- Putting code into the memory (no zeros)
- Getting %eip to point to our code (distance buff to stored eip)
- Finding the return address (guess the raw address)

How can we make this more difficult

Defense: Canaries in the Stack

- Run time tests for stack integrity.

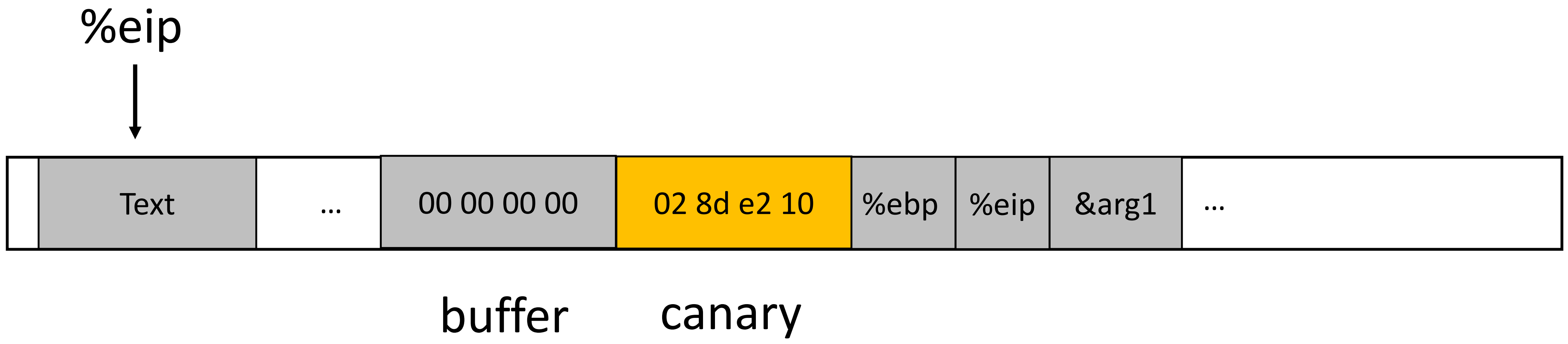
- Embed “canaries” in stack frames and verify their integrity prior to function return.



Defense: Canaries in the Stack

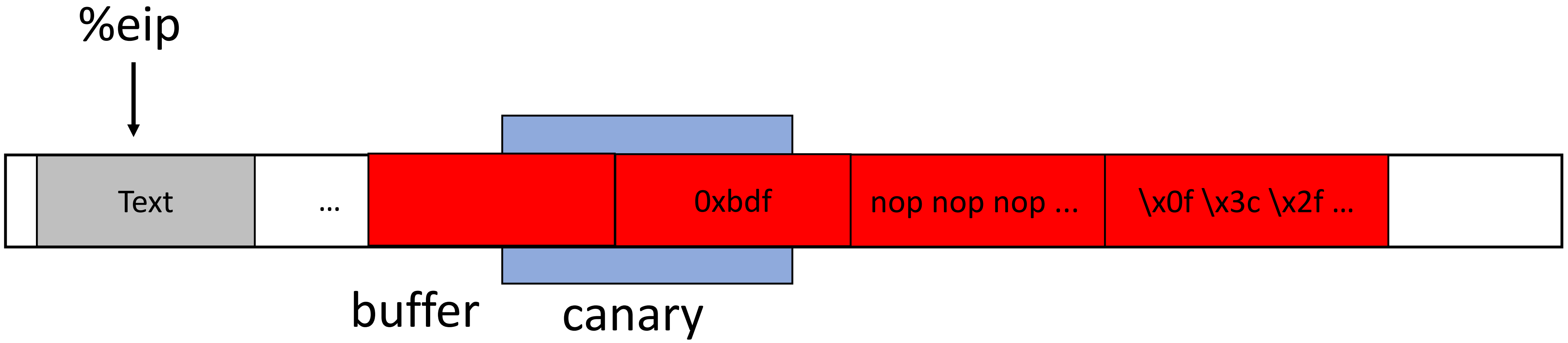
- Compiler inserts some extra code in function
  - At entry: after pushing ebp, push an unlikely bit pattern (canary)
  - At return: before popping ebp, check the canary
    - Terminate program if canary has changed
- Counter-attack: rewrite the canary in overflow
  - guess the canary
  - read the canary (via, eg, printf vulnerabilities)

Detecting Overflows with Canaries



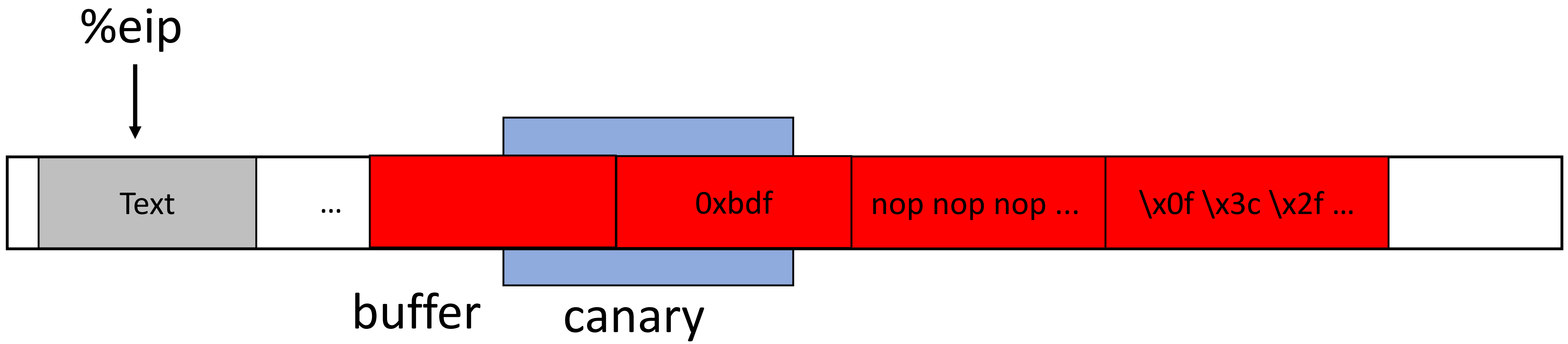
Detecting Overflows with Canaries

Abort due to unexpected value



Detecting Overflows with Canaries

Abort due to unexpected value



What should we use as canary values?

Detecting Overflows with Canaries

- Random canary:
  - Random string chosen at program startup.
  - Insert canary string into every stack frame.
  - Verify canary before returning from function.
    - Exit program if canary changed. Turns potential exploit into DoS.
  - To corrupt the stack, attacker must learn current random string.

- Terminator canary: Canary = {0, newline, linefeed, EOF}
  - String functions will not copy beyond terminator.
  - Attacker cannot use string functions to corrupt the stack.

Canary Values

- Terminator canaries (CR, LF, NULL, -1)
  - Leverages the fact that scanf etc. don't allow these

From StackGuard [Wagle & Cowan]

- Random canaries
  - Write a new random value @ each process start
  - Save the real value somewhere in memory
  - Must write-protect the stored value

- Random XOR canaries
  - Same as random canaries
  - But store canary XOR some control info, instead (random canary + eip value)

StackGuard

- StackGuard implemented as a GCC patch
    - Program must be recompiled

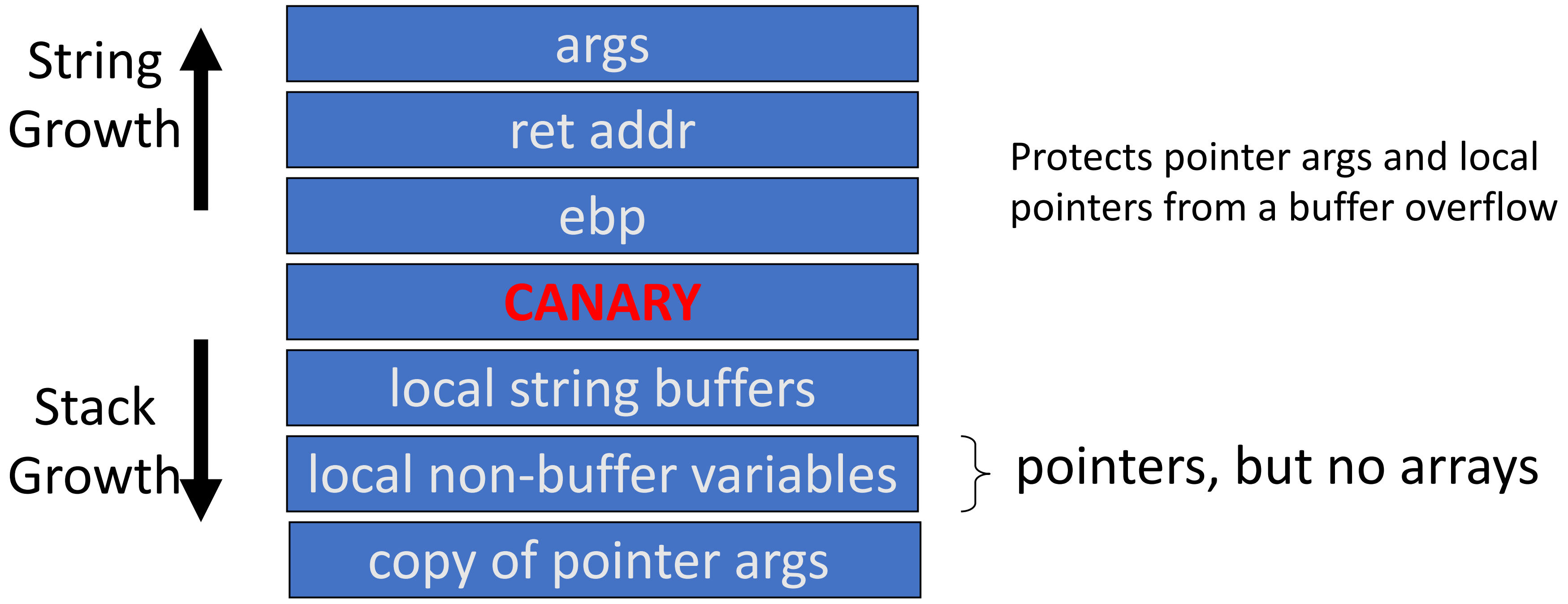
- Minimal performance effects: 8% for Apache

- Note: Canaries do not provide full protection
    - Some stack smashing attacks leave canaries unchanged

- Heap protection: PointGuard
    - Protects function pointers by encrypting them: e.g. XOR with random cookie
    - Less effective, more noticeable performance effects

StackGuard Enhancements: ProPolice

- ProPolice (IBM) - gcc 3.4.1. (**-fstack-protector**)
  - Rearrange stack layout to prevent ptr overflow.



Summary: Canaries are not full proof

- Canaries are an important defense tool, but do not prevent all control hijacking attacks:
  - Heap-based attacks still possible
  - Integer overflow attacks still possible
  - /GS by itself does not prevent Exception Handling attacks

    (also need SAFESEH and SEHOP)

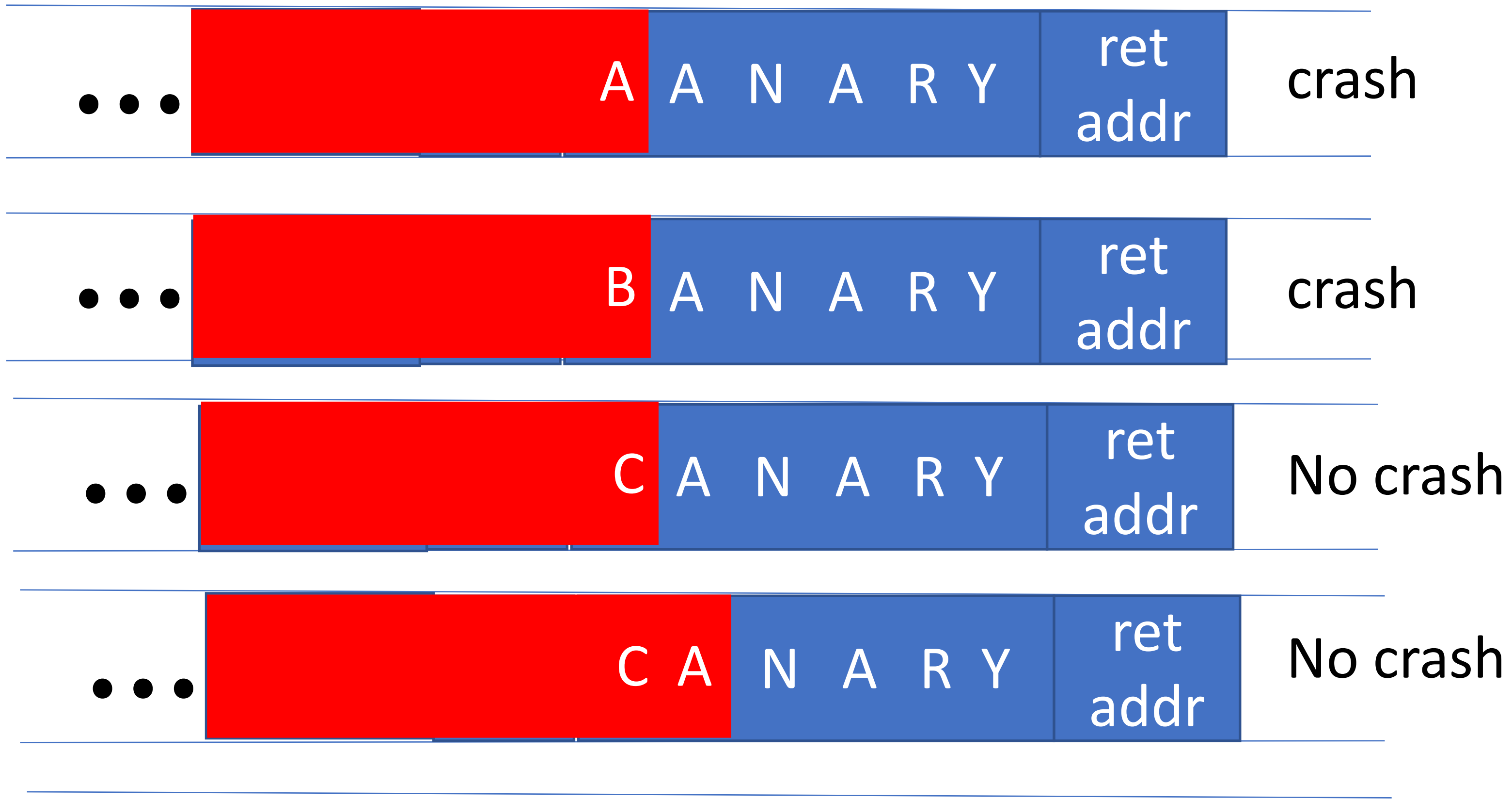
Even worse: canary extraction

A common design for crash recovery:

- When process crashes, restart automatically (for availability)
- Often canary is unchanged (reason: relaunch using fork)

Danger:

- canary extraction byte by byte



- What is the main problem with code injection onto the stack?

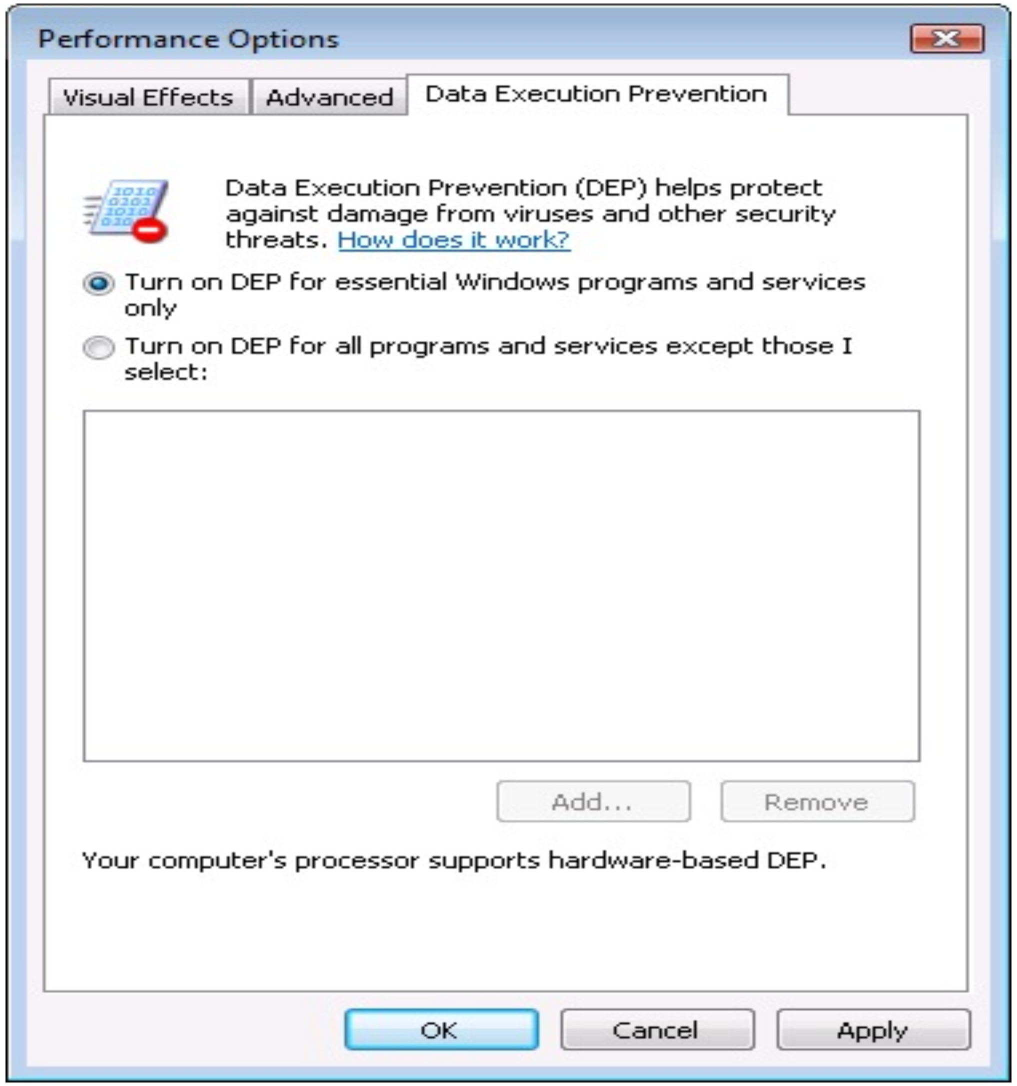
Defense: Marking Memory as Non-Exec.

Prevent attack code execution by marking stack and heap as **non-executable**

Attempt to execute stack results in exception (data should never be exec.)

- NX-bit on AMD Athlon 64, XD-bit on Intel P4 Prescott
  - NX bit in every Page Table Entry (PTE)
- Deployment:
  - Linux (via PaX project); OpenBSD
  - Windows: since XP SP2 (DEP)
    - Visual Studio: **/NXCompat[:NO]**
- Limitations:
  - Some apps need executable heap (e.g. JITs).
  - Can be easily bypassed using **Return Oriented Programming (ROP)**

Examples: DEP controls in Windows



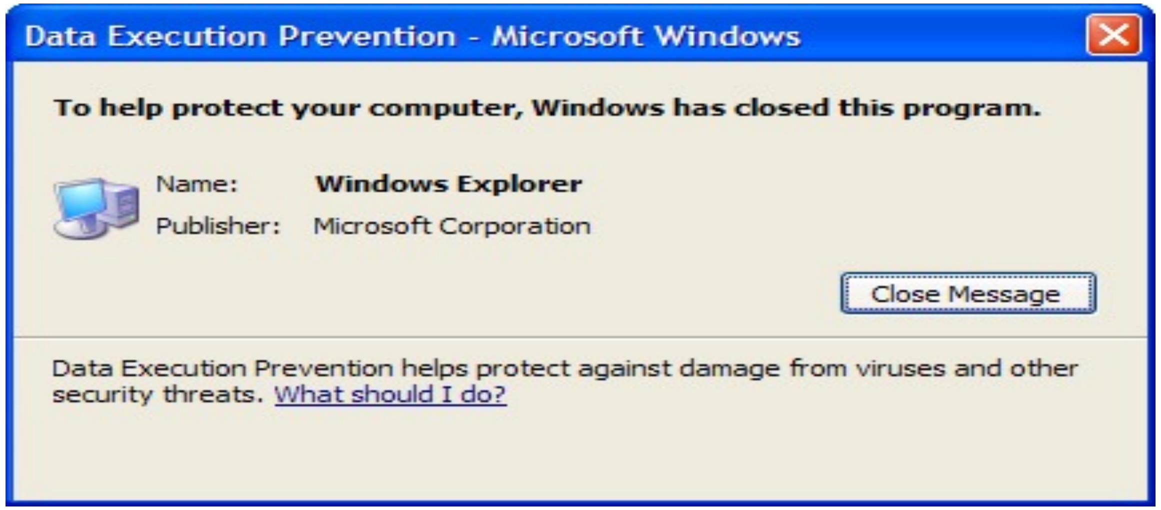
Performance Options

Data Execution Prevention (DEP) helps protect against damage from viruses and other security threats. How does it work?

[✓ Turn on DEP for essential Windows programs and services only]

Turn on DEP for all programs and services except those I select:

Your computer's processor supports hardware-based DEP.



Data Execution Prevention - Microsoft Windows

To help protect your computer, Windows has closed this program.

Name: **Windows Explorer**

Publisher: Microsoft Corporation

Data Execution Prevention helps protect against damage from viruses and other security threats. What should I do?

DEP terminating a program

Challenges Revisited…

- Putting code into the memory (no zeros)
    - Option: Make this detectable with canaries

- Getting %eip to point to our code (dist buff to stored eip)
    - Non-executable stack (not bullet proof)

- Finding the return address (guess the raw address)

How can we make this more difficult

Defense: Randomization

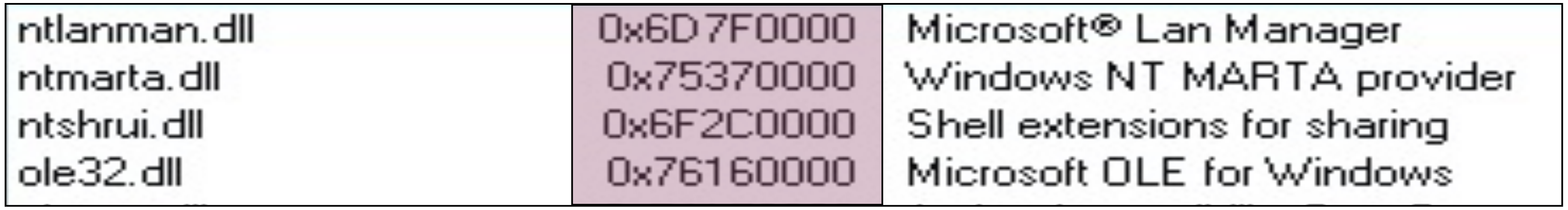
- **ASLR**: (Address Space Layout Randomization)
    - Change the layout of the stack
    - Map shared libraries to rand location in process memory

      ⇒ Attacker cannot jump directly to exec function

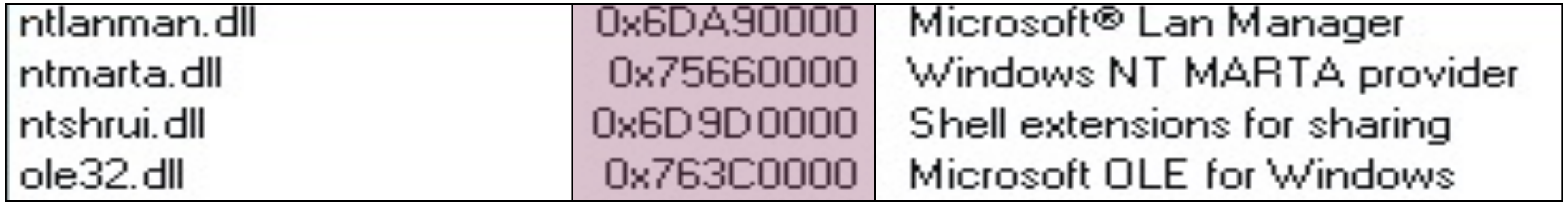
- Deployment: (/DynamicBase)
    - **Windows 7**: 8 bits of randomness for DLLs
        - aligned to 64K page in a 16MB region ⇒ 256 choices
    - **Windows 8:** 24 bits of randomness on 64-bit processors

ASLR Example

Booting twice loads libraries into different locations:



ntlanman.dll	0x6D7F0000	Microsoft® Lan Manager
ntmarta.dll	0x75370000	Windows NT MARTA provider
ntshrui.dll	0x6F2C0000	Shell extensions for sharing
ole32.dll	0x76160000	Microsoft OLE for Windows



ntlanman.dll	0x6DA90000	Microsoft® Lan Manager
ntmarta.dll	0x75660000	Windows NT MARTA provider
ntshrui.dll	0x6D9D0000	Shell extensions for sharing
ole32.dll	0x763C0000	Microsoft OLE for Windows

Note: everything in process memory must be randomized

stack, heap, shared libs, base image

- Win 8 **Force ASLR**: ensures all loaded modules use ASLR

Slow ASLR Adoption

- ASLR has been adopted slowly in the industry
- Linux in 2005
- Vista in 2007 (off by default for compatibility with older software)
- OS X in 2007 (for system libraries), 2011 for all apps
- iOS 4.3 (2011)
- Android 4.0
- FreeBSD: no

Defenses vs. Attack Responses

- Defense: Make stack/heap non-executable to prevent injection of code
    - Attack response: Return to libc
- Defense: Hide the address of desired libc code or return address using ASLR
    - Attack response: Brute force search (for 32-bit systems)
    - Other exploits involve prefetch attacks (64-bit systems)

Defenses vs. Attack Responses

- Defense: Avoid using libc code entirely and use code in the program text instead

    - Attack response: Construct needed functionality using return oriented programming (ROP)

Return Oriented Programming (ROP)

Return-oriented Programming

- Introduced by Hovav Shacham in 2007
- The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86), CCS'07
- Idea: rather than use a single (libc) function to run your shellcode, string together pieces of existing code, called gadgets, to do it instead
- Challenges
  - Find the gadgets you need
  - String them together

Approach

- Gadgets are instruction groups that end with ret

- Stack serves as the code

    - %esp = program counter

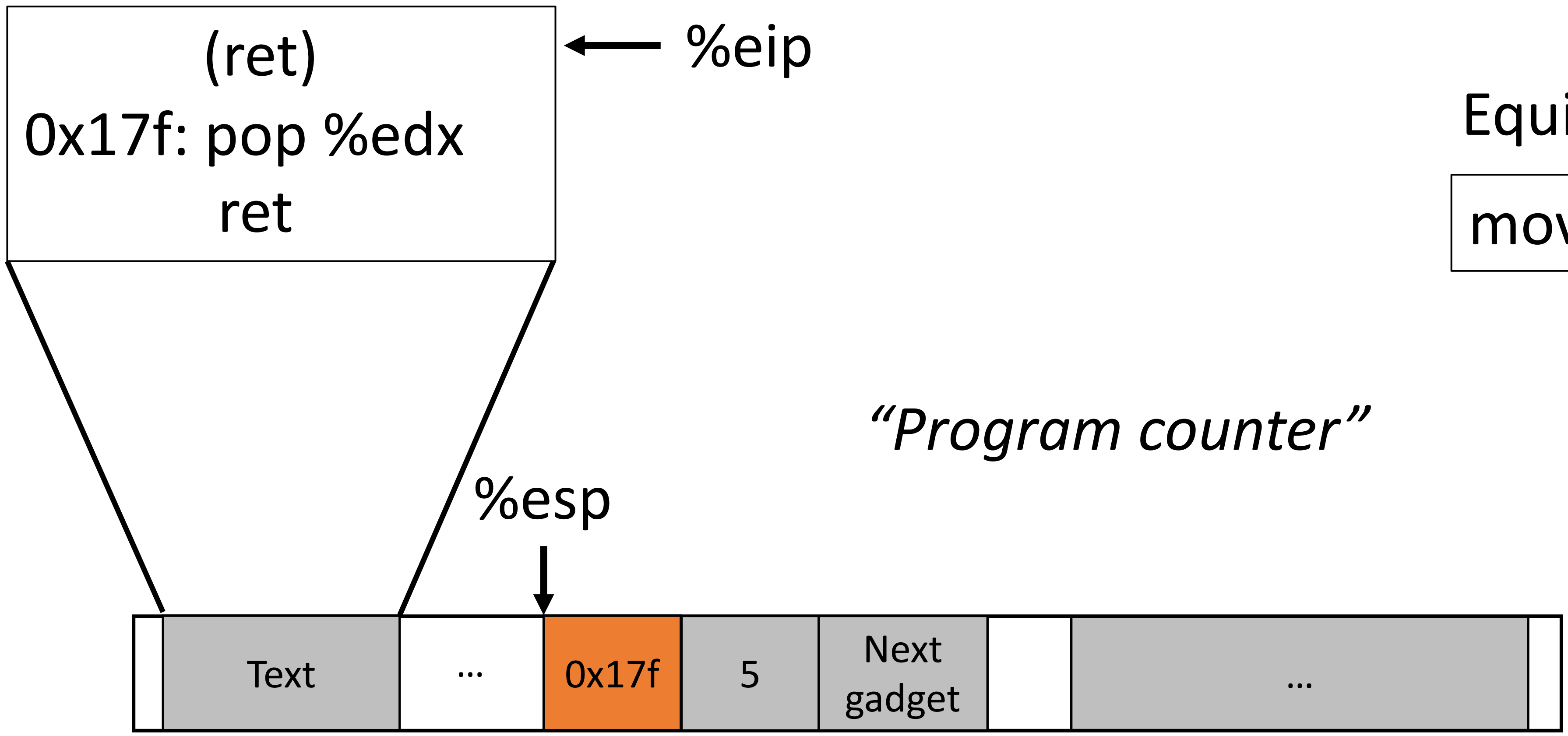
    - Gadgets invoked via ret instruction

    - Gadgets get their arguments via pop, etc.
        - Also on the stack

Finding the gadgets?

- How can we find gadgets to construct an exploit?
  - Automate a search of the target binary for gadgets (look for ret instructions, work backwards)
    - https://github.com/0vercl0k/rp
- Are there sufficient gadgets to do anything interesting?
  - Yes: Shacham found that for significant codebases (e.g., libc), gadgets are Turing complete
    - Especially true on x86's dense instruction set
- Schwartz et al (USENIX Security '11) have automated gadget shellcode creation, though not needing/requiring Turing completeness

Simple Example



Equivalent to

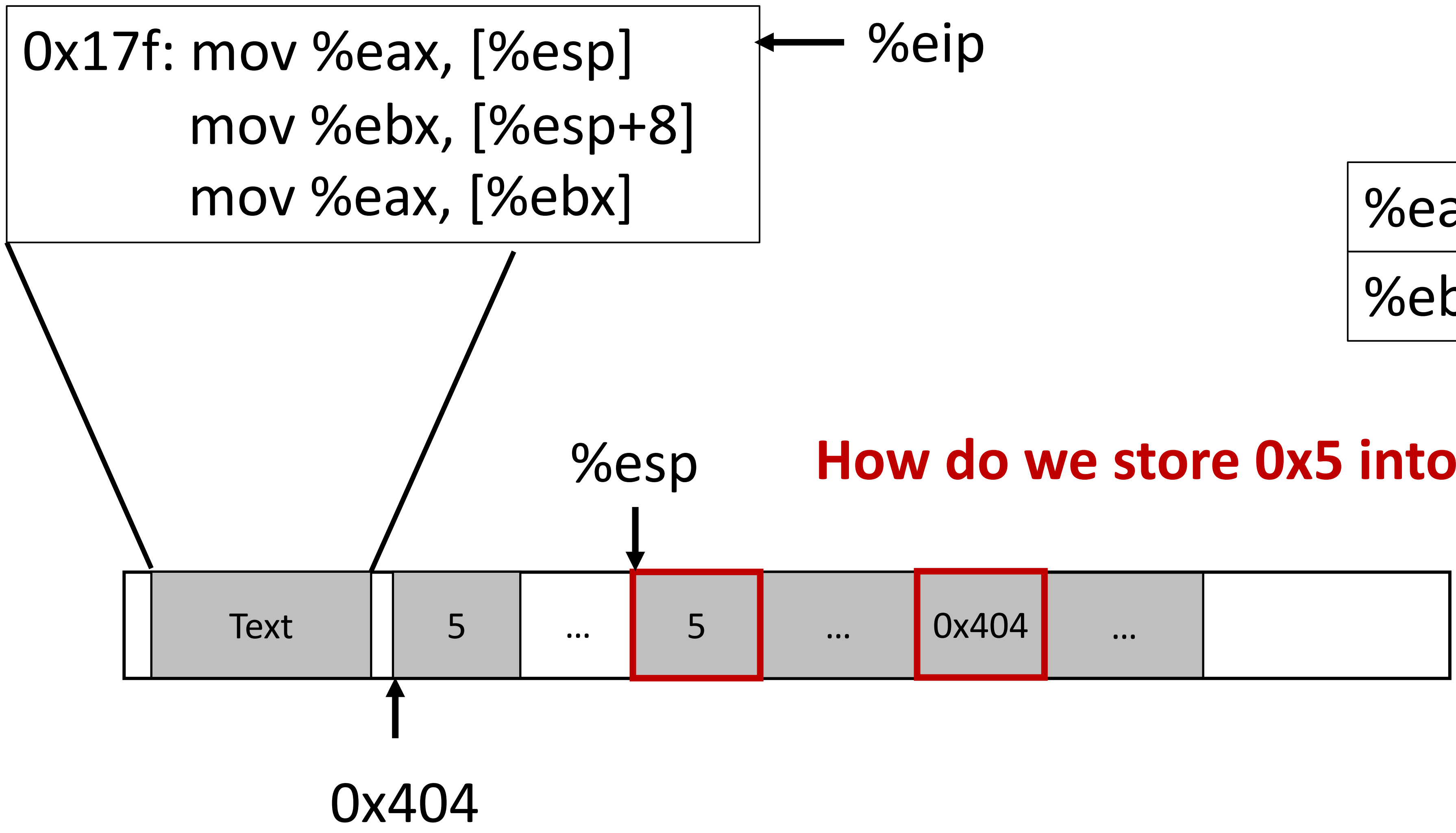
mov %edx, 5

"Program counter"

"Instructions"

Code Sequence

```
0x17f: mov %eax, [%esp]
       mov %ebx, [%esp+8]
       mov %eax, [%ebx]
```

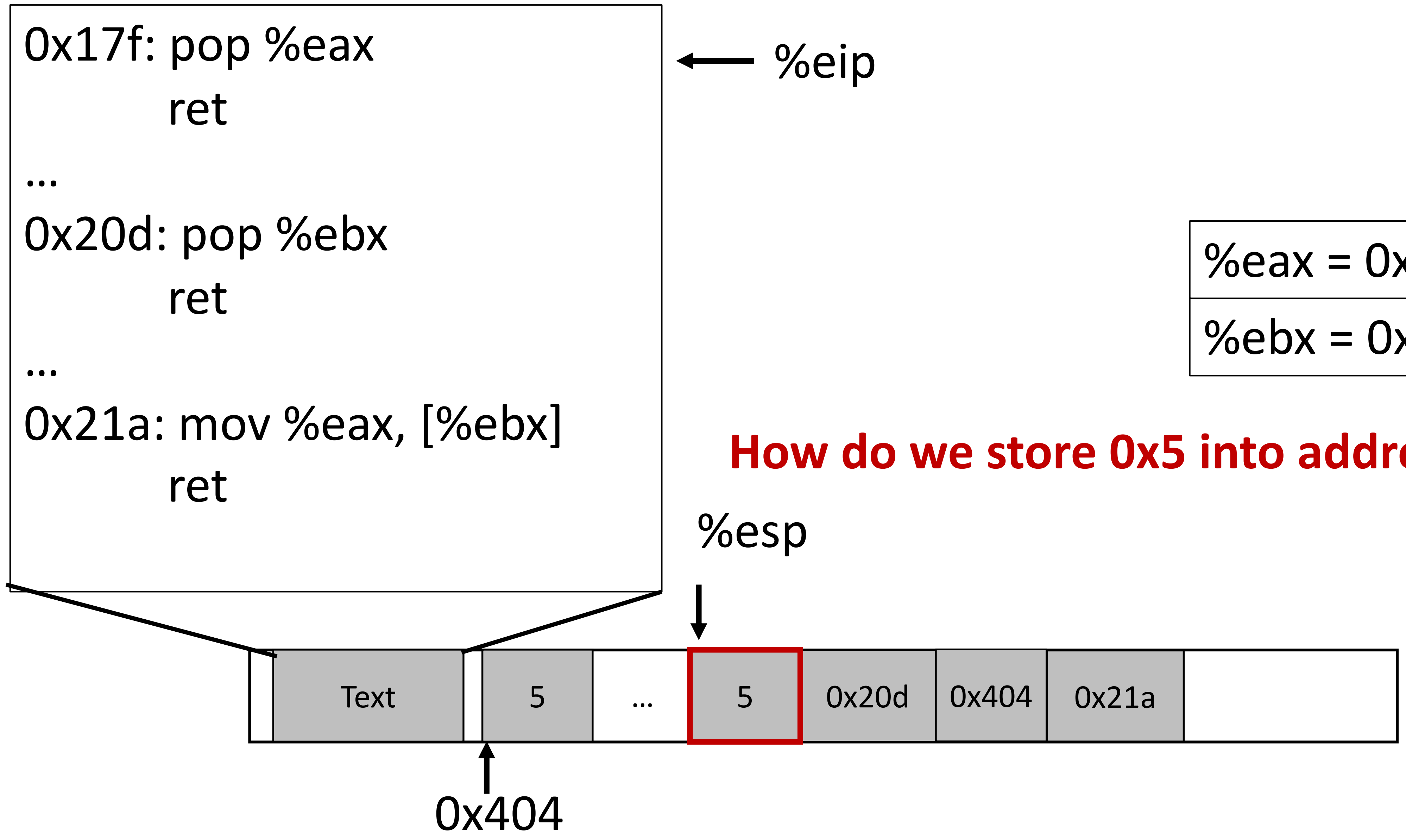


%eax = 0x5
%ebx = 0x404

How do we store 0x5 into address 0x404?

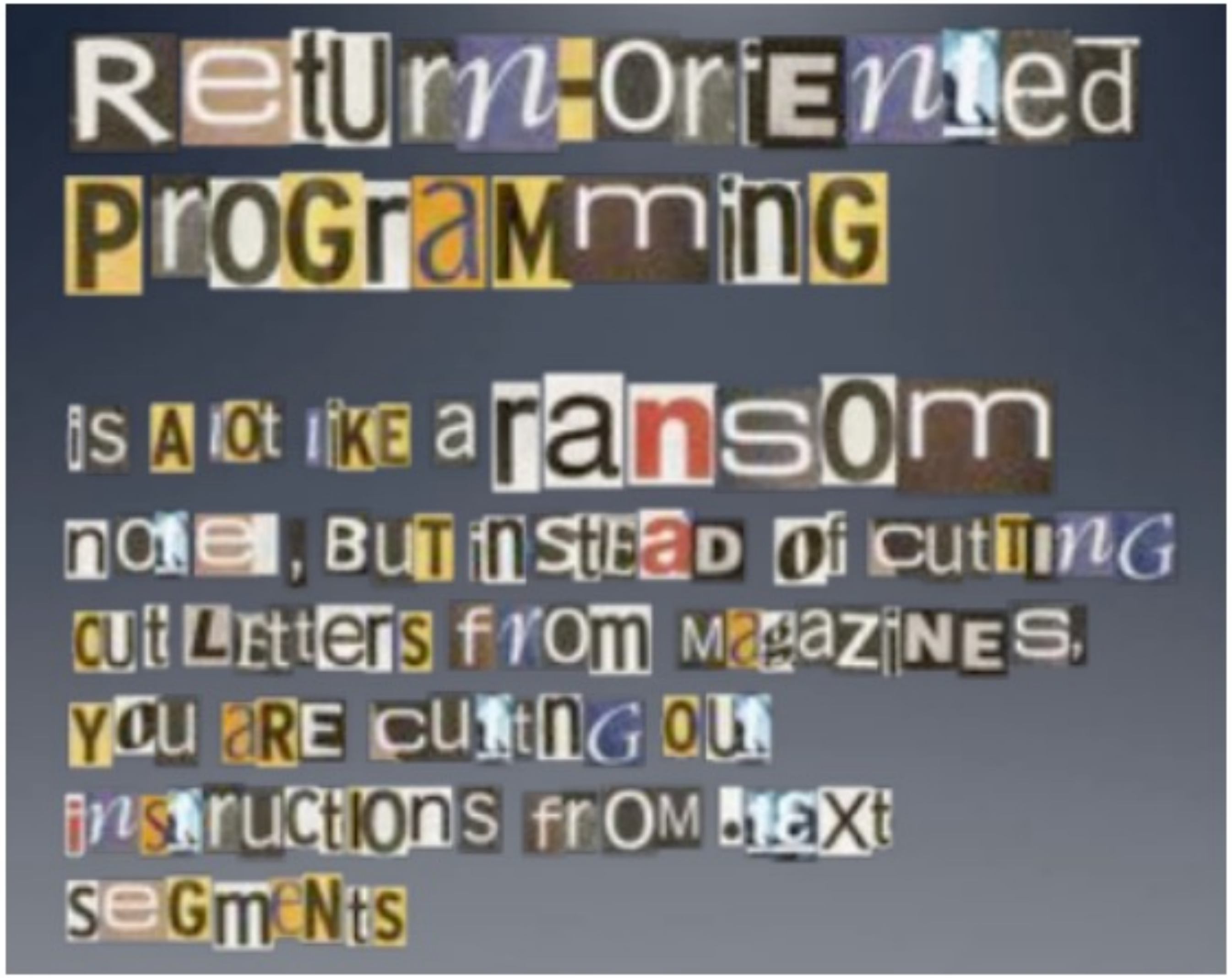
Equivalent ROP Sequence

```
0x17f: pop %eax
       ret
…
0x20d: pop %ebx
       ret
…
0x21a: mov %eax, [%ebx]
       ret
```



%eax = 0x5
%ebx = 0x404

How do we store 0x5 into address 0x404?


Return-Oriented Programming

is a lot like a ransom note, but instead of cutting cut letters from magazines, you are cutting out instructions from text segments

Defensive Coding for Memory Safety

Defensive Coding Practices

- Think defensive driving
  - Avoid depending on anyone else around you
  - If someone does something unexpected, you won't crash (or worse)
  - It's about **minimizing trust**
- Each module takes responsibility for checking the validity of all inputs sent to it
  - Even if you "know" your callers will never send a NULL pointer…
  - …Better to throw an exception (or even exit) than run malicious code

http://nob.cs.ucdavis.edu/bishop/secprog/robust.html

Secure coding practices

```
char digit_to_char(int i) {
    char convert[] = “0123456789”;
    return convert[i];
}
```

Think about all potential inputs, no matter how peculiar

Secure coding practices

```
char digit_to_char(int i) {
    char convert[] = “0123456789”;
    return convert[i];
}
```

Think about all potential inputs, no matter how peculiar

```
char digit_to_char(int i) {
    char convert[] = “0123456789”;
    if(i < 0 || i > 9)
        return ‘?’;
    return convert[i];
}
```

Enforce rule compliance at runtime

Automated Testing

How to program defensively

- Code reviews, real or imagined
  - Organize your code so it is obviously correct
  - Re-write until it would be self-evident to a reviewer

“Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.”

- Remove the opportunity for programmer mistakes with better languages and libraries
  - Java performs automatic bounds checking
  - C++ provides a safe std::string class

Automated Testing Techniques

- Static code analysis
  - Detects most bugs
  - Not automatable: model checking or theorem proving
- Dynamic code analysis
  - Monitor execution (in a vm?) for memory safety: valgrind, address-sanitizer
  - But only checks those executions
  - High overhead: not suitable for deployed code
- Penetration testing
  - actively generate inputs to exploit vulnerabilities
  - applicable to programs, applications, network, servers
  - Fuzz testing: many many random inputs

What Happens Once You Find an Issue

- Try to find the root cause
- Is there a smaller input that crashes in the same spot? (Make it easier to understand)
- Are there multiple crashes that point back to the same bug?
- Determine if this crash represents an exploitable vulnerability
- In particular, is there a buffer overrun?

Finding Memory Errors

- Compile the program with Address Sanitizer (ASAN)
  - Instruments accesses to arrays to check for overflows, and use-after-free errors

  https://code.google.com/p/address-sanitizer/

- Fuzz it

- Did the program crash with an ASAN-signaled error? Then worry about exploitability

- Similarly, you can compile with other sorts of error checkers for the purposes of testing
  - E.g., valgrind memcheck http://valgrind.org/

END

Backup

Formatted I/O

- Recall: C’s printf family of functions

- Format specifiers, list of arguments
  - Specifier indicates type of argument (%s, %i, etc.)
  - Position in string indicates argument to print

```
void print_record(int age, char *name)
{
   printf(“Name: %s\tAge: %d\n”,name,age);
}
```

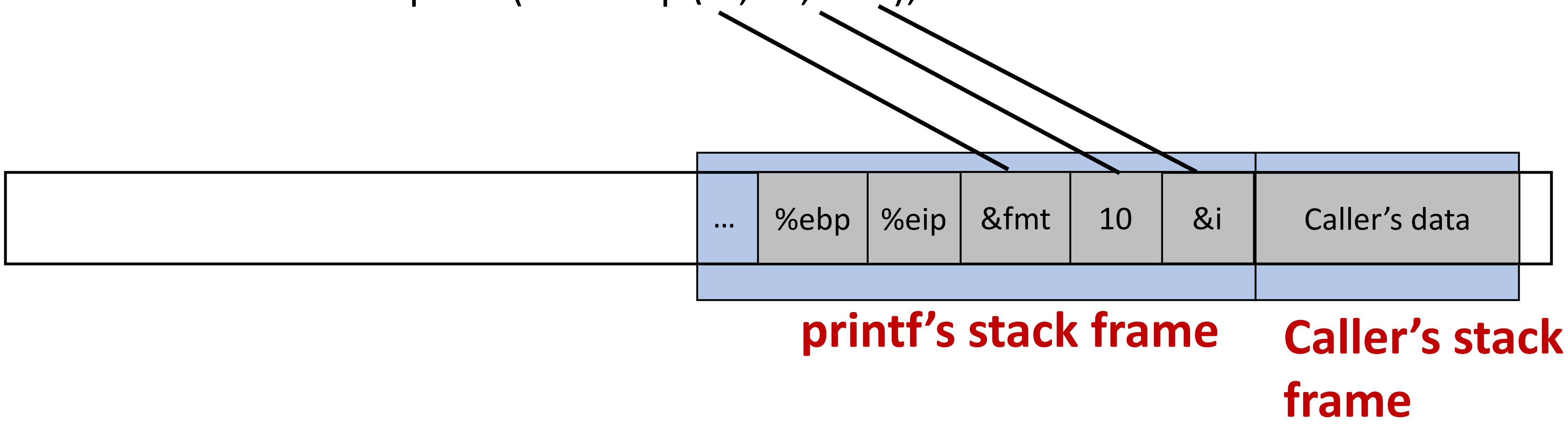
What’s the Difference?

```
void vulnerable()
{
   char buf[80];
   if(fgets(buf, sizeof(buf), stdin)==NULL)
         return;
   printf(buf);
}

void safe()
{
    char buf[80];
    if(fgets(buf, sizeof(buf), stdin)==NULL)
         return;
    printf(“%s”,buf);
}
```

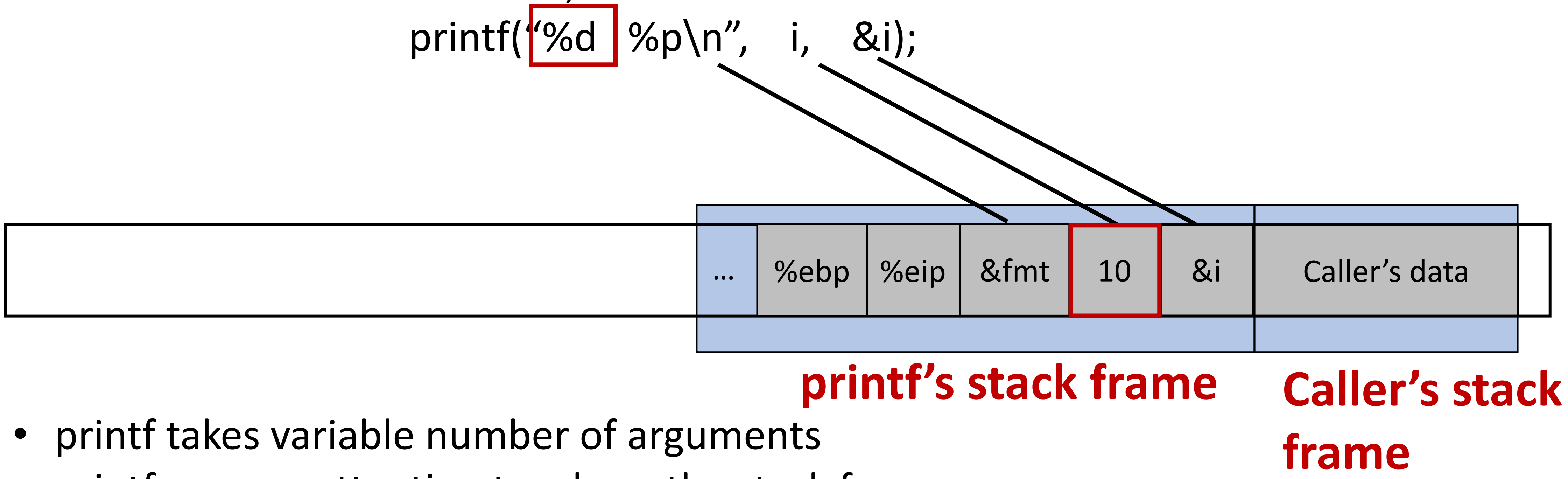
printf Format Strings

```
int i = 10;
printf("%d   %p\n",   i,   &i);
```



printf Format Strings

```
int i = 10;
printf("%d  %p\n",   i,   &i);
```

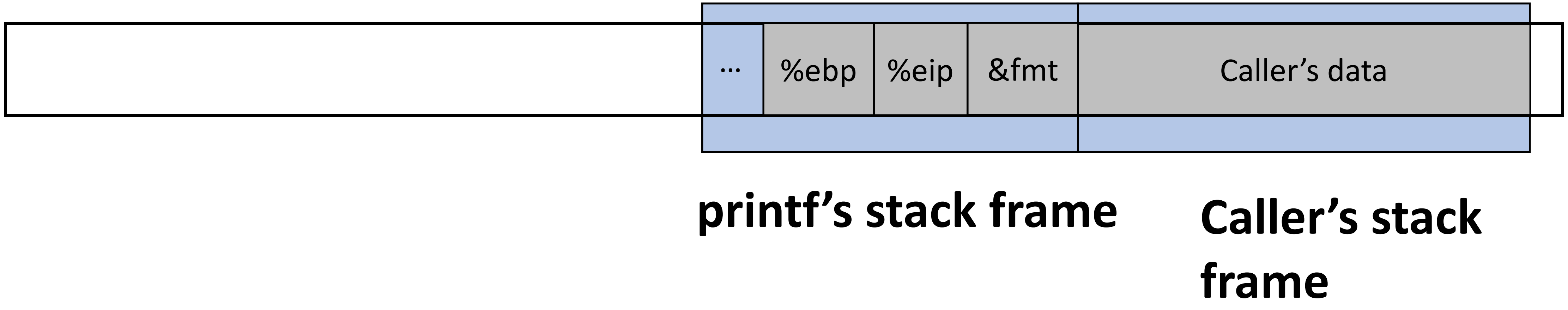


- printf takes variable number of arguments
- printf pays no attention to where the stack frame "ends"
- It presumes that you called it with (at least) as many arguments as specified in the format string

Vulnerable Program

```
void vulnerable()
{
    char buf[80];
    if(fgets(buf, sizeof(buf), stdin)==NULL)
        return;
    printf(buf);
}
```

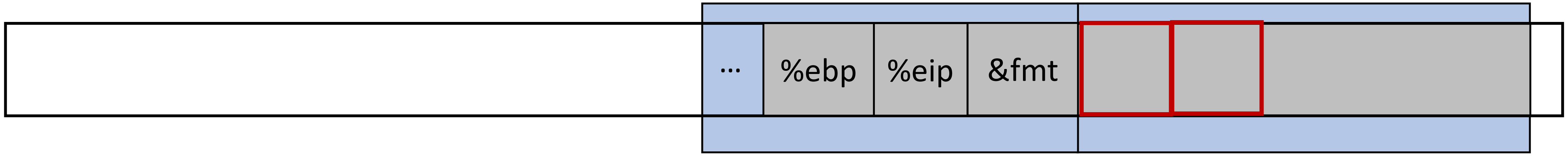
Why is this program vulnerable?



Vulnerable Program

```
void vulnerable()
{
    char buf[80];
    if(fgets(buf, sizeof(buf), stdin)==NULL)
        return;
    printf(buf);
}
```

%d %x



printf's stack frame

Caller's stack frame

Exercise

- Design a format string that can print a secret value stored on the stack located 20 bytes away from the pointer to the format string?

Modifying Memory

- printf() has the ability to overwrite a variable

- printf(“Hello %n “, &i);
  - %n will count the number of bytes printed so far and store them into the address of &i

- We can use this to modify data on the stack!

Format string vulnerabilities

- printf(%s);
- printf(%d %d %d %d …);
- printf(“%08x %08x %08x %08x …”);
- printf(“100% no way!”)

Format string vulnerabilities

- printf(%s);
  - Prints bytes pointed to by that stack entry
- printf(“%d %d %d %d …”);

- printf(“%08x %08x %08x %08x …”);

- printf(“100% no way!”)

Format string vulnerabilities

- printf(%s);
  - Prints bytes pointed to by that stack entry
- printf(“%d %d %d %d …”);
  - Prints a series of stack entries as integers
- printf(“%08x %08x %08x %08x …”);

- printf(“100% no way!”)

Format string vulnerabilities

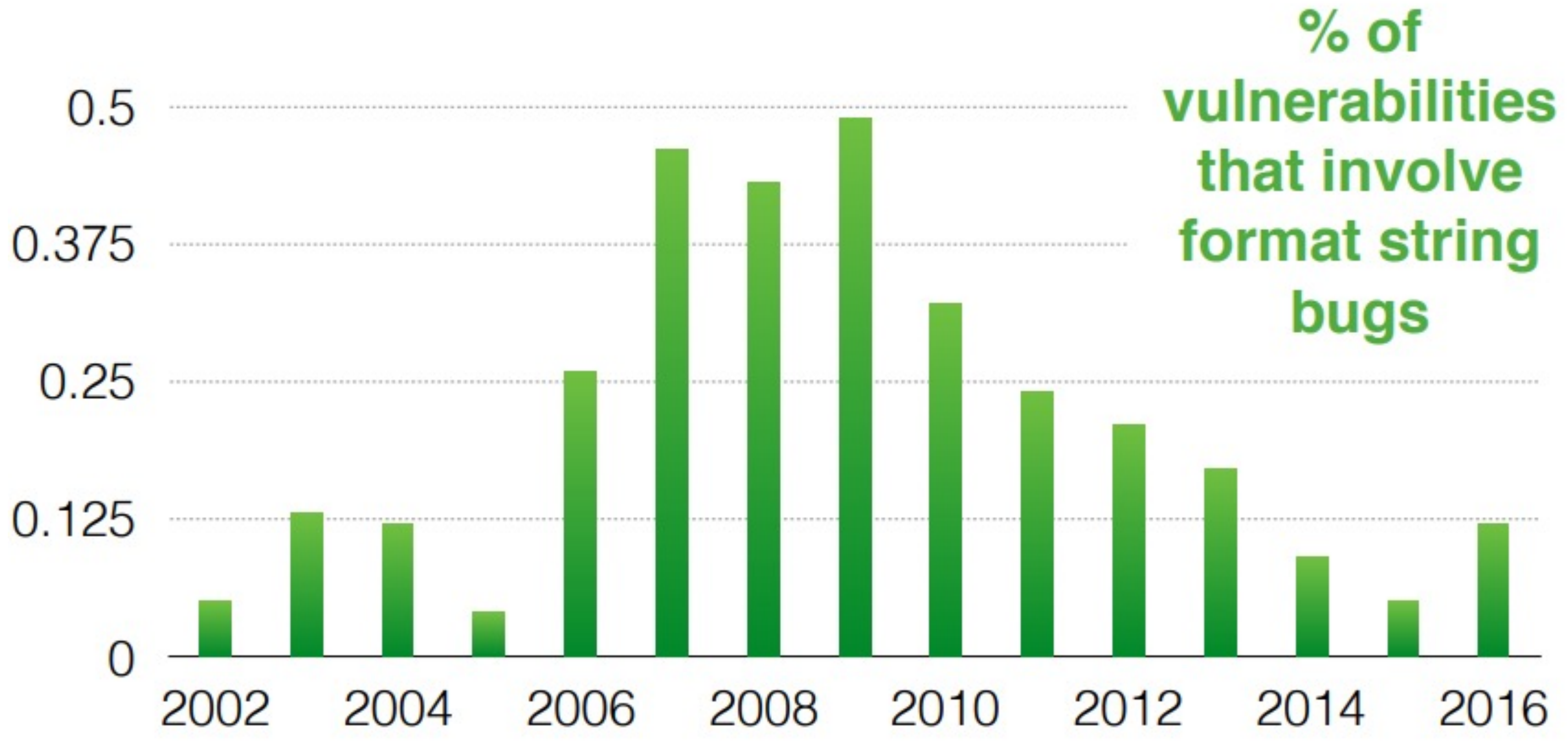
- printf(%s);
  - Prints bytes pointed to by that stack entry
- printf(“%d %d %d %d …”);
  - Prints a series of stack entries as integers
- printf(“%08x %08x %08x %08x …”);
  - Same, but nicely formatted hex
- printf(“100% no way!”)

Other Format string vulnerabilities

- Dumping arbitrary memory:
  - Walk up stack until desired pointer is found.
  - printf(“%08x.%08x.%08x.%08x|%s|”)

- Writing to arbitrary memory:
  - printf(“hello %n”, &temp)   -- writes ‘6’ into temp.
  - printf(“%08x.%08x.%08x.%08x.%n”)

Format string Prevalence



https://nvd.nist.gov/view/vuln/statistics

MS Visual Studio /GS (Since 2003)

Compiler /GS option:

- Combination of ProPolice and Random canary.
- If cookie mismatch, default behavior is to call **_exit(3)**

Function prolog:

```
    sub   esp, 8     // allocate 8 bytes for cookie
    mov   eax, DWORD PTR ___security_cookie
    xor   eax, esp      // xor cookie with current esp
    mov   DWORD PTR [esp+8], eax  // save in stack
```

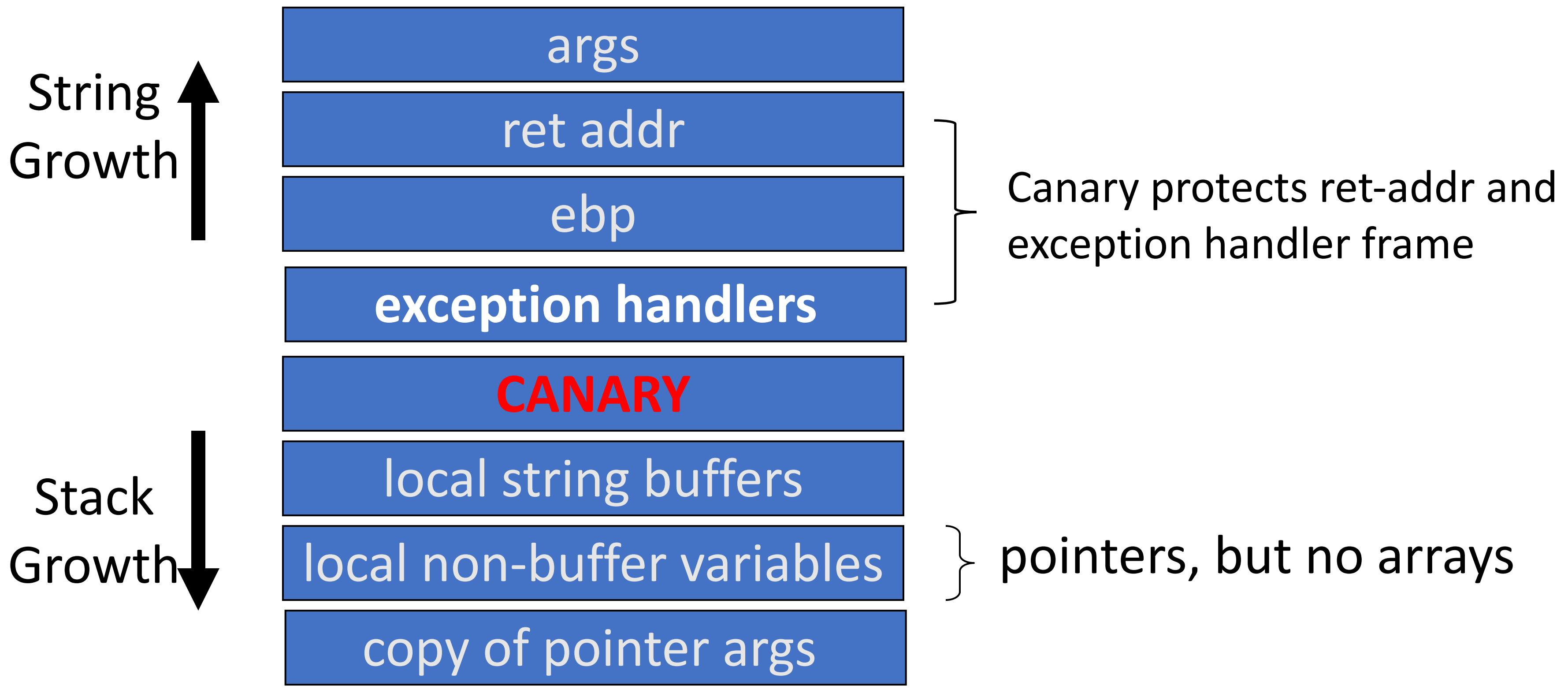
Function epilog:

```
    mov   ecx, DWORD PTR  [esp+8]
    xor   ecx, esp
    call  @__security_check_cookie@4
    add   esp, 8
```

Enhanced /GS in Visual Studio 2010:

– /GS protection added to all functions, unless can be proven unnecessary

MS Visual Studio /GS (Since 2003)

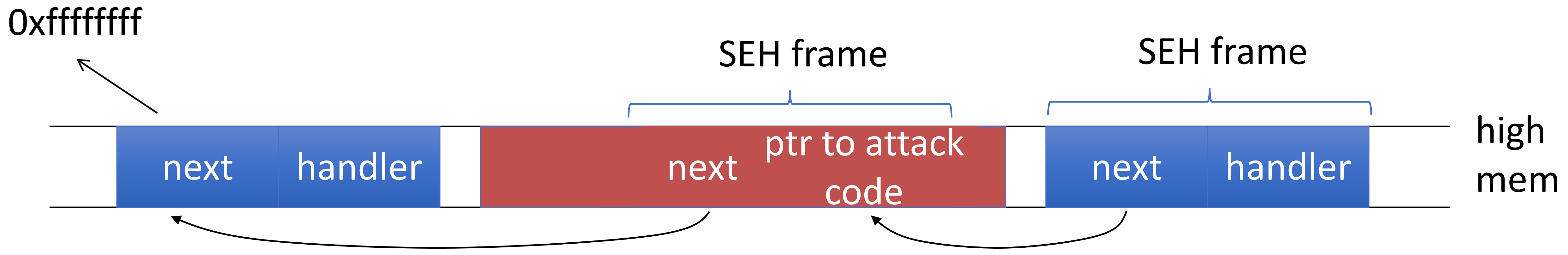


Evading /GS with exception handlers

- When exception is thrown, dispatcher walks up exception list until handler is found (else use default handler)

After overflow: handler points to attacker's code

exception triggered ⇒ control hijack



Defenses: SAFESEH and SEHOP

- /SAFESEH: linker flag (Structured Exception Handling)
  - Linker produces a binary with a table of safe exception handlers
  - System will not jump to exception handler not on list

- /SEHOP: platform defense (Structured Exception Handling Overwrite Protection - since win vista SP1)
  - Observation: SEH attacks typically corrupt the "next" entry in SEH list.
  - SEHOP: add a dummy record at top of SEH list
  - When exception occurs, dispatcher walks up list and verifies dummy record is there. If not, terminates process.