CSE 484 / CSE M 584: Computer Security and Privacy

Software Security: Buffer Overflow Attacks

(continued)

Spring 2015

Franziska (Franzi) Roesner
franzi@cs.washington.edu

Thanks to Dan Boneh, Dieter Gollmann, Dan Halperin, Yoshi Kohno, John Manferdelli, John Mitchell, Vitaly Shmatikov, Bennet Yee, and many others for sample slides and materials ...

Looking Forward

- Today: More buffer overflows + defenses
- Wednesday: one more day on software security
- Friday: guest lecture by David Aucsmith
- Next week: start crypto

- Ethics form due **Wednesday**
- Homework #1 due April 17
- Lab #1 out this week (please form groups!)

- Section this week: Lab 1

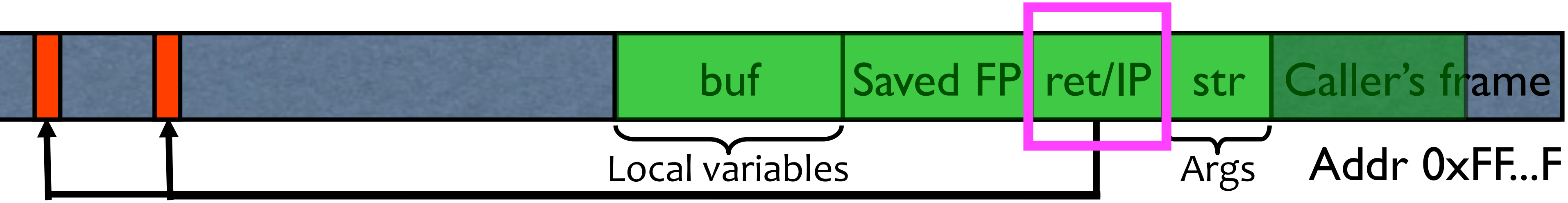
Last Time: Basic Buffer Overflows

- Memory pointed to by str is copied onto stack…

```
void func(char *str) {
        char buf[126];
        strcpy(buf,str);
   }
```

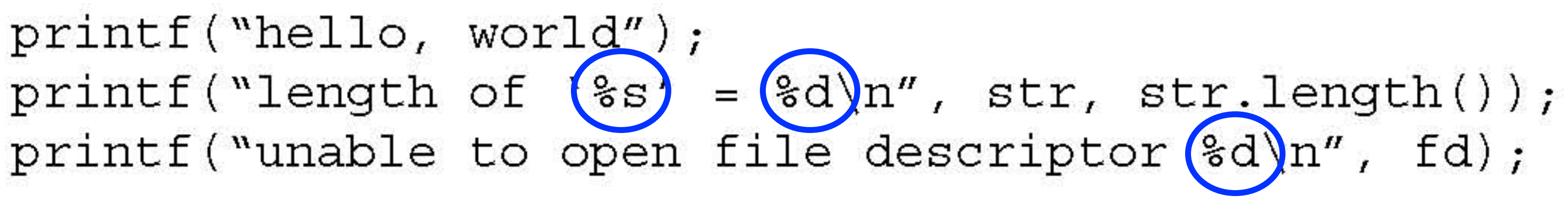
strcpy does NOT check whether the string at *str contains fewer than 126 characters

- If a string longer than 126 bytes is copied into buffer, it will overwrite adjacent stack locations.


This will be interpreted as return address!

Variable Arguments in C

- In C, can define a function with a variable number of arguments
  - Example: void printf(const char* format, …)
- Examples of usage:



```
printf(“hello, world”);
printf(“length of ‘%s’ = %d\n”, str, str.length());
printf(“unable to open file descriptor %d\n”, fd);
```

Format specification encoded by special % characters

%d,%i,%o,%u,%x,%X – integer argument
%s – string argument
%p – pointer argument (void *)
Several others

Format Strings in C

- Proper use of printf format string:

```
int foo = 1234;
printf("foo = %d in decimal, %X in hex",foo,foo);
```

This will print:

```
foo = 1234 in decimal, 4D2 in hex
```

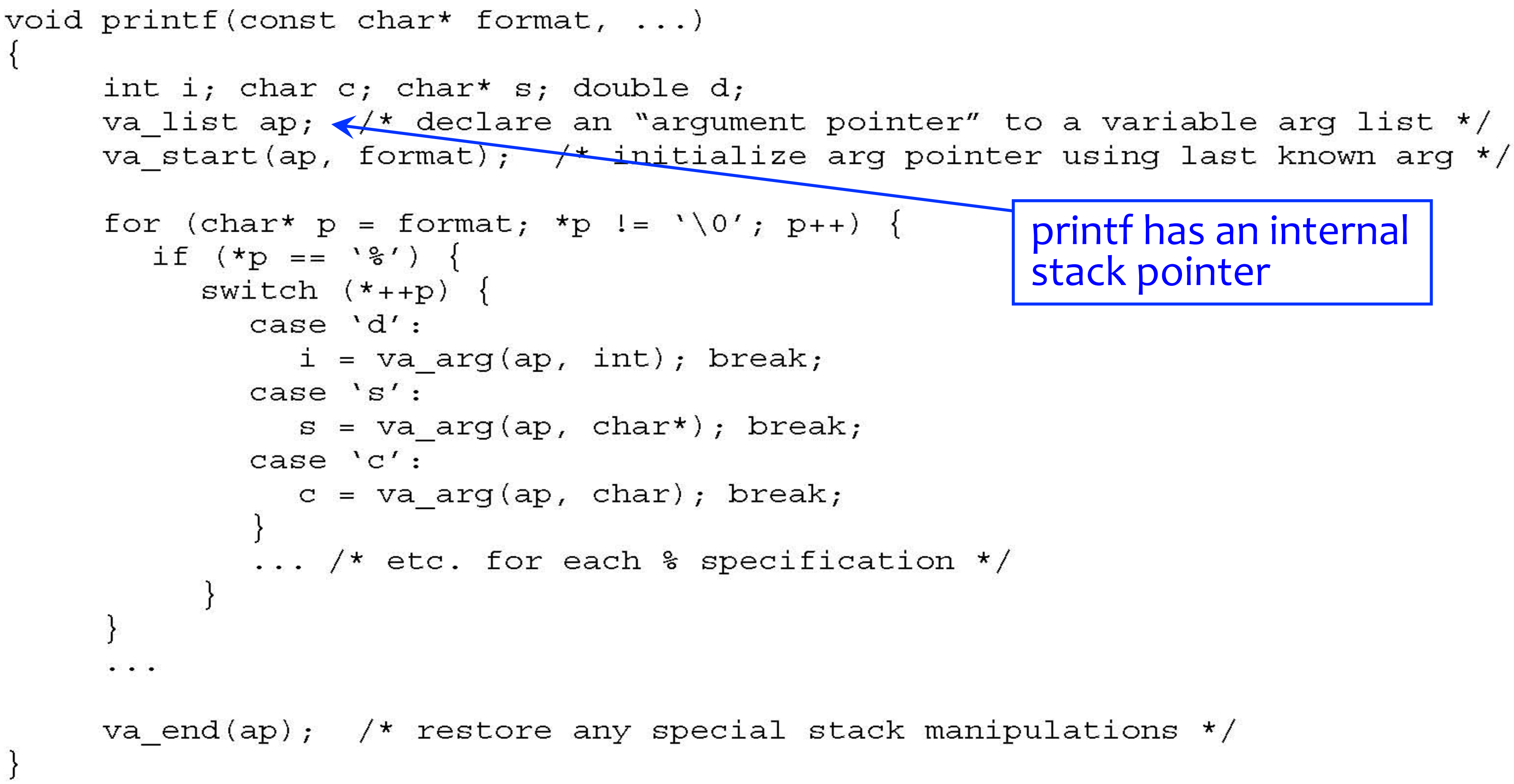
- Sloppy use of printf format string:

```
char buf[14] = "Hello, world!";
printf(buf);
// should've used printf("%s", buf);
```

What happens if buffer contains format symbols starting with % ???

Implementation of Variable Args

- Special functions va_start, va_arg, va_end compute arguments at run-time



```
void printf(const char* format, ...)
{
      int i; char c; char* s; double d;
      va_list ap; /* declare an “argument pointer” to a variable arg list */
      va_start(ap, format);  /* initialize arg pointer using last known arg */

      for (char* p = format; *p != ‘\0’; p++) {
         if (*p == ‘%’) {
            switch (*++p) {
               case ‘d’:
                  i = va_arg(ap, int); break;
               case ‘s’:
                  s = va_arg(ap, char*); break;
               case ‘c’:
                  c = va_arg(ap, char); break;
               }
               ... /* etc. for each % specification */
            }
      }
      ...

      va_end(ap);  /* restore any special stack manipulations */
}
```

printf has an internal stack pointer

Format Strings in C

If the buffer contains format symbols starting with %, the location pointed to by printf’s internal stack pointer will be interpreted as an argument of printf.

This can be exploited to move printf’s internal stack pointer!

- Sloppy use of printf format string:

```
char buf[14] = “Hello, world!”;
printf(buf);
// should’ve used printf(“%s”, buf);
```

What happens if buffer contains format symbols starting with % ???

Viewing Memory

- %x format symbol tells printf to output data on stack

```
printf("Here is an int:  %x",i);
```

- What if printf does <u>not</u> have an argument?

```
char buf[16]="Here is an int:  %x";
printf(buf);
```

- Or what about:

```
char buf[16]="Here is a string:  %s";
printf(buf);
```

Viewing Memory

- %x format symbol tells printf to output data on stack

```
printf("Here is an int:  %x",i);
```

- What if printf does not have an argument?

```
char buf[16]="Here is an int:  %x";
printf(buf);
```

  - Stack location pointed to by printf's internal stack pointer will be interpreted as an int. (What if crypto key, password, ...?)

- Or what about:

```
char buf[16]="Here is a string:  %s";
printf(buf);
```

  - Stack location pointed to by printf's internal stack pointer will be interpreted as a pointer to a string

Writing Stack with Format Strings

- %n format symbol tells printf to write the number of characters that have been printed

```
printf("Overflow this!%n",&myVar);
```

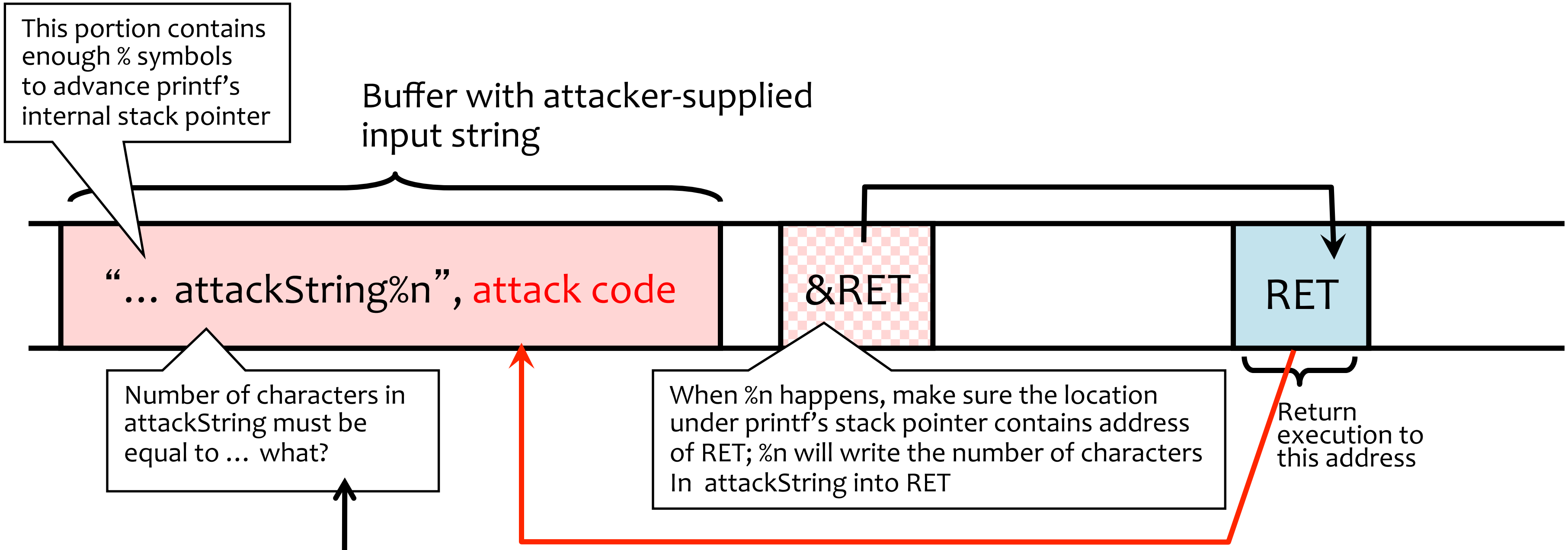
  - Argument of printf is interpeted as destination address
  - This writes 14 into myVar ("Overflow this!" has 14 characters)

- What if printf does not have an argument?

```
char buf[16]="Overflow this!%n";
printf(buf);
```

  - Stack location pointed to by printf's internal stack pointer will be **interpreted as address** into which the number of characters will be written.

Using %n to Overwrite Return Address



This portion contains enough % symbols to advance printf’s internal stack pointer

Buffer with attacker-supplied input string

Number of characters in attackString must be equal to … what?

When %n happens, make sure the location under printf’s stack pointer contains address of RET; %n will write the number of characters In attackString into RET

Return execution to this address

C allows you to concisely specify the “width” to print, causing printf to pad by printing additional blank characters without reading anything else off the stack.

Example: printf(“%5d”, 10) will print three spaces followed by the integer: “   10”

That is, %n will print 5, not 2.

Key idea: do this 4 times with the right numbers to overwrite the return address byte-by-byte.
(4x %n to write into &RET, &RET+1, &RET+2, &RET+3)

Recommended Reading

- It will be hard to do Lab 1 without reading:
  - Smashing the Stack for Fun and Profit
  - Exploiting Format String Vulnerabilities
- Links to these readings are posted on the course schedule.

Buffer Overflow: Causes and Cures

- Typical memory exploit involves code injection
  - Put malicious code at a predictable location in memory, usually masquerading as data
  - Trick vulnerable program into passing control to it

- We'll talk about a few defenses today:
  1. Prevent execution of untrusted code
  2. Stack "canaries"
  3. Encrypt pointers
  4. Address space layout randomization

W⊕X / DEP

- Mark all writeable memory locations as non-executable
  - Example: Microsoft's Data Execution Prevention (DEP)
  - This blocks (almost) all code injection exploits
- Hardware support
  - AMD "NX" bit, Intel "XD" bit (in post-2004 CPUs)
  - Makes memory page non-executable
- Widely deployed
  - Windows (since XP SP2),
    Linux (via PaX patches),
    OS X (since 10.5)

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?

What Does W⊕X Not Prevent?

- Can still corrupt stack …
  - … or function pointers or critical data on the heap
- As long as "saved EIP" points into existing code, W⊕X protection will not block control transfer
- This is the basis of return-to-libc exploits
  - Overwrite saved EIP with address of any library routine, arrange stack to look like arguments
- Does not look like a huge threat
  - Attacker cannot execute arbitrary code, especially if system() is not available

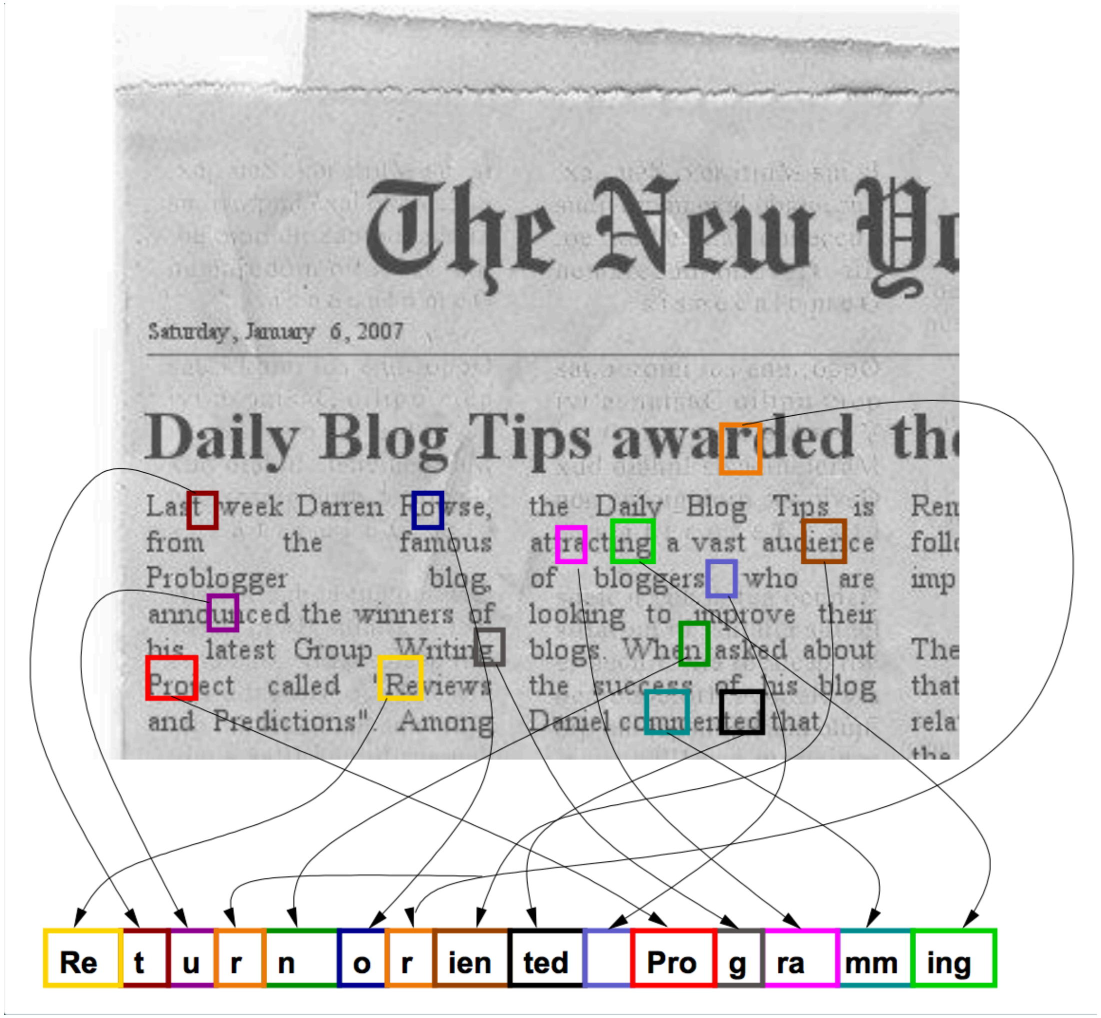
return-to-libc on Steroids

- Overwritten saved EIP need not point to the beginning of a library routine
- **Any** existing instruction in the code image is fine
  - Will execute the sequence starting from this instruction
- What if instruction sequence contains RET?
  - Execution will be transferred… to where?
  - Read the word pointed to by stack pointer (ESP)
    - Guess what? Its value is under attacker's control!
  - Use it as the new value for EIP
    - Now control is transferred to an address of attacker's choice!
  - Increment ESP to point to the next word on the stack

Chaining RETs for Fun and Profit

- Can chain together sequences ending in RET
  - Krahmer, "x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique" (2005)
- What is this good for?
- Answer [Shacham et al.]: everything
  - Turing-complete language
  - Build "gadgets" for load-store, arithmetic, logic, control flow, system calls
  - Attack can perform arbitrary computation using no injected code at all – return-oriented programming

Return-Oriented Programming



The New Y

Saturday, January 6, 2007

Daily Blog Tips awarded th

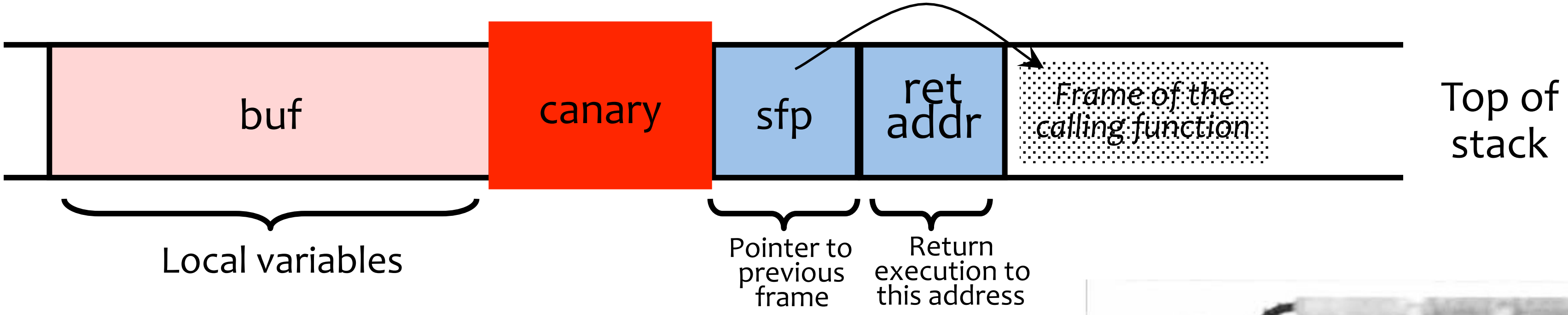
Last week Darren Rowse, from the famous Problogger blog, announced the winners of his latest Group Writing Project called 'Reviews and Predictions". Among the Daily Blog Tips is attracting a vast audience of bloggers who are looking to improve their blogs. When asked about the success of his blog Daniel commented that

Other Issues with W⊕X / DEP

- Some applications require executable stack
  - Example: Flash ActionScript, Lisp, other interpreters
- Some applications are not linked with /NXcompat
  - DEP disabled (e.g., some Web browsers)
- JVM makes all its memory RWX – readable, writable, executable
  - Inject attack code over memory containing Java objects, pass control to them
- "Return" into a memory mapping routine, make page containing attack code writeable

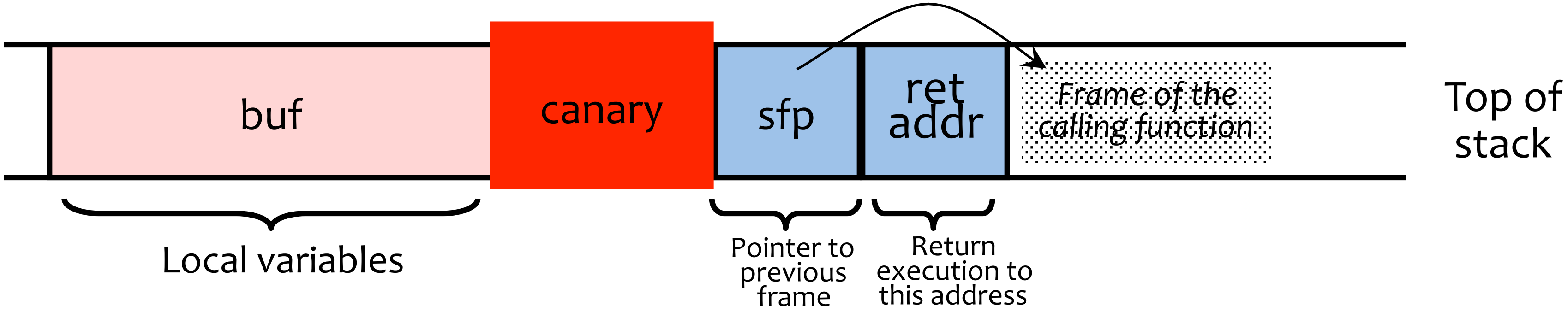
Run-Time Checking: StackGuard

- Embed "canaries" (stack cookies) in stack frames and verify their integrity prior to function return
  - Any overflow of local variables will damage the canary



Run-Time Checking: StackGuard

- Embed "canaries" (stack cookies) in stack frames and verify their integrity prior to function return
  - Any overflow of local variables will damage the canary



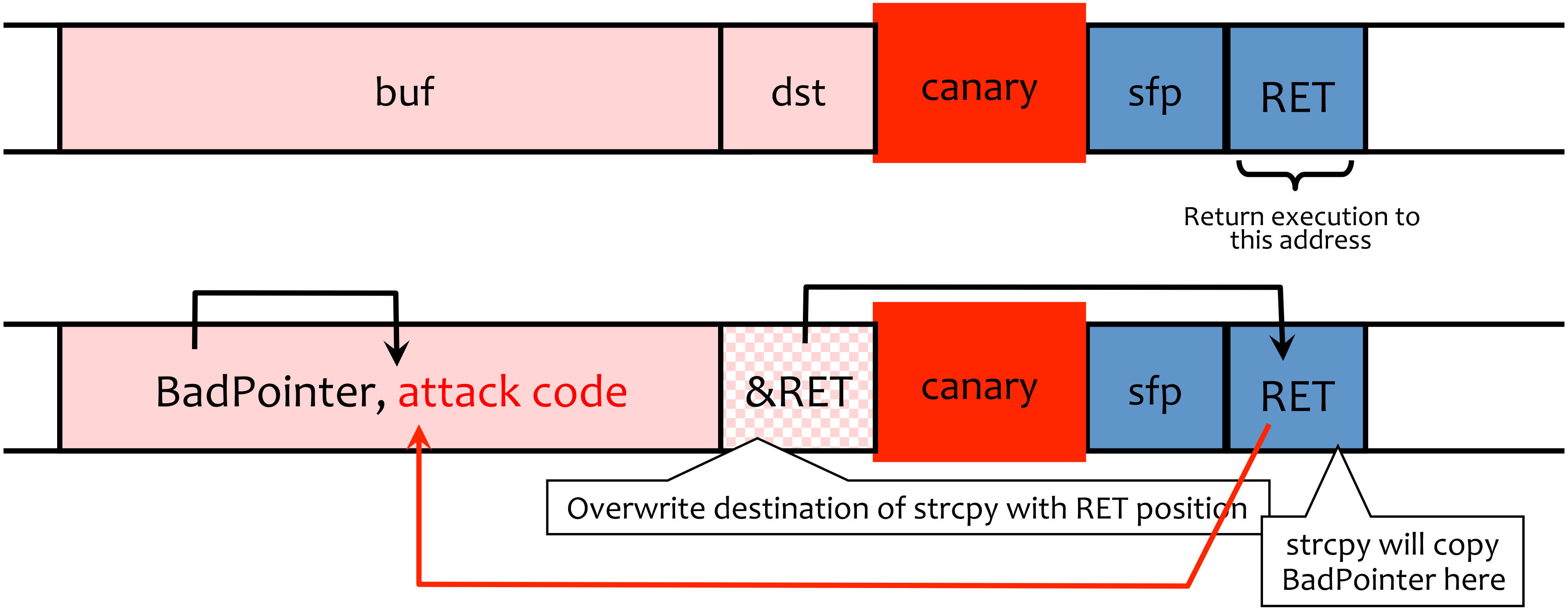
- Choose random canary string on program start
  - Attacker can't guess what the value of canary will be
- Terminator canary: "\0", newline, linefeed, EOF
  - String functions like strcpy won't copy beyond "\0"

StackGuard Implementation

- StackGuard requires code recompilation
- Checking canary integrity prior to every function return causes a performance penalty
  - For example, 8% for Apache Web server
- StackGuard can be defeated
  - A single memory write where the attacker controls both the value and the destination is sufficient

Defeating StackGuard

- Suppose program contains strcpy(dst,buf) where attacker controls both dst and buf
  - Example: dst is a local pointer variable

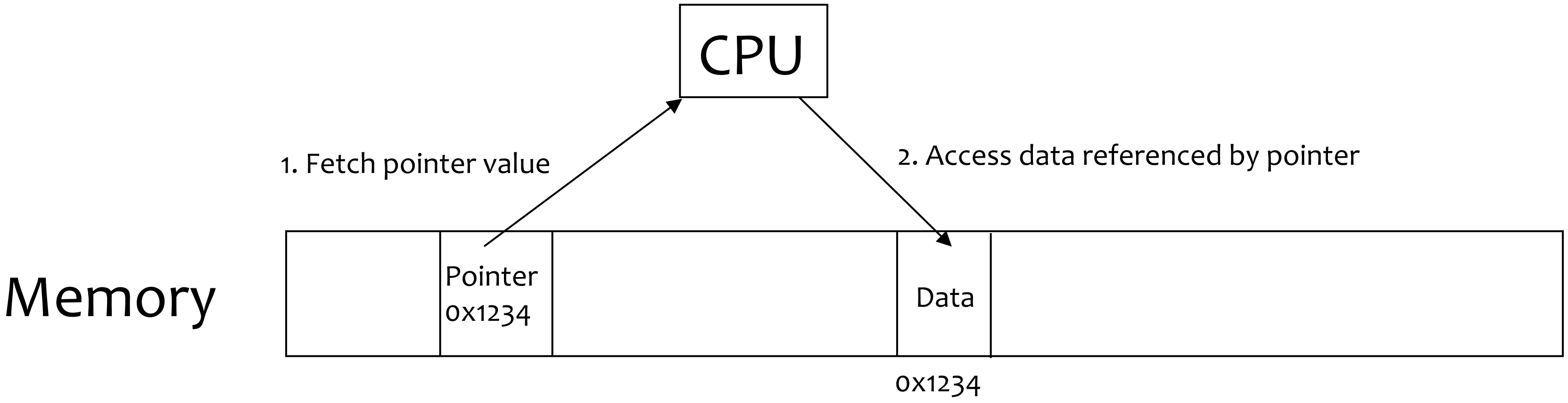


PointGuard

- Attack: overflow a function pointer so that it points to attack code
- Idea: encrypt all pointers while in memory
  - Generate a random key when program is executed
  - Each pointer is XORed with this key when loaded from memory to registers or stored back into memory
    - Pointers cannot be overflowed while in registers
- Attacker cannot predict the target program's key
  - Even if pointer is overwritten, after XORing with key it will dereference to a "random" memory address

[Cowan]

Normal Pointer Dereference



CPU

1. Fetch pointer value

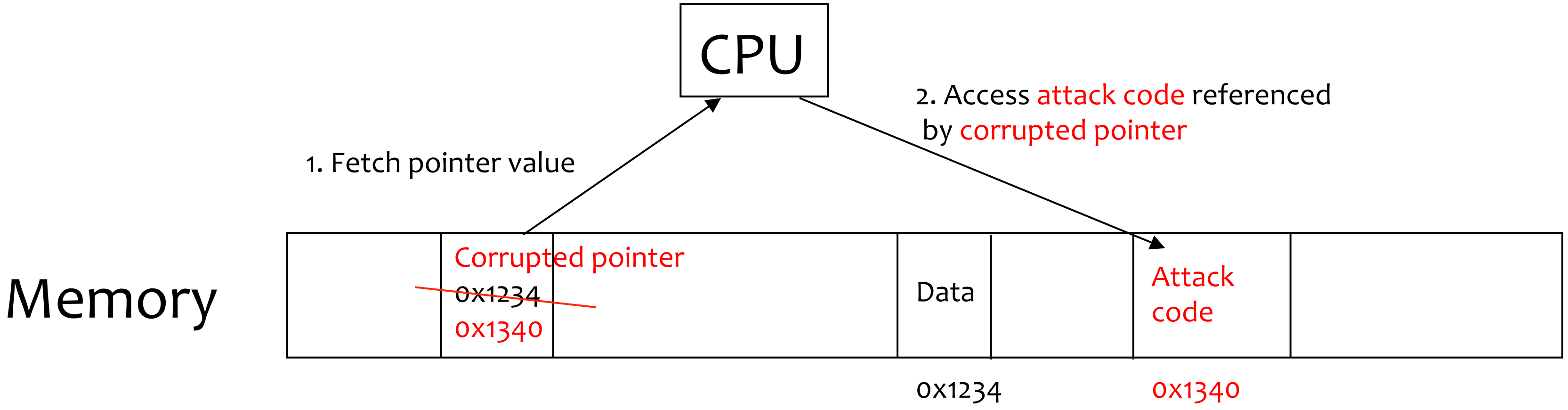
2. Access data referenced by pointer

Memory

Pointer
0x1234

Data

0x1234



CPU

1. Fetch pointer value

2. Access attack code referenced
by corrupted pointer

Memory

Corrupted pointer
~~0x1234~~
0x1340

Data

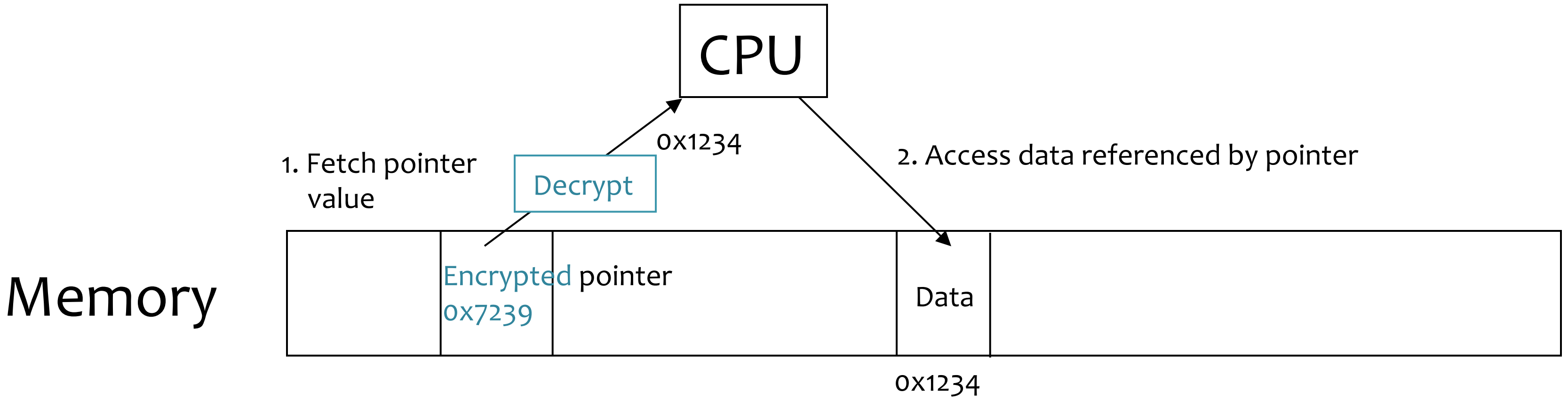
Attack
code

0x1234

0x1340

[Cowan]

PointGuard Dereference



CPU

0x1234

1. Fetch pointer value

Decrypt

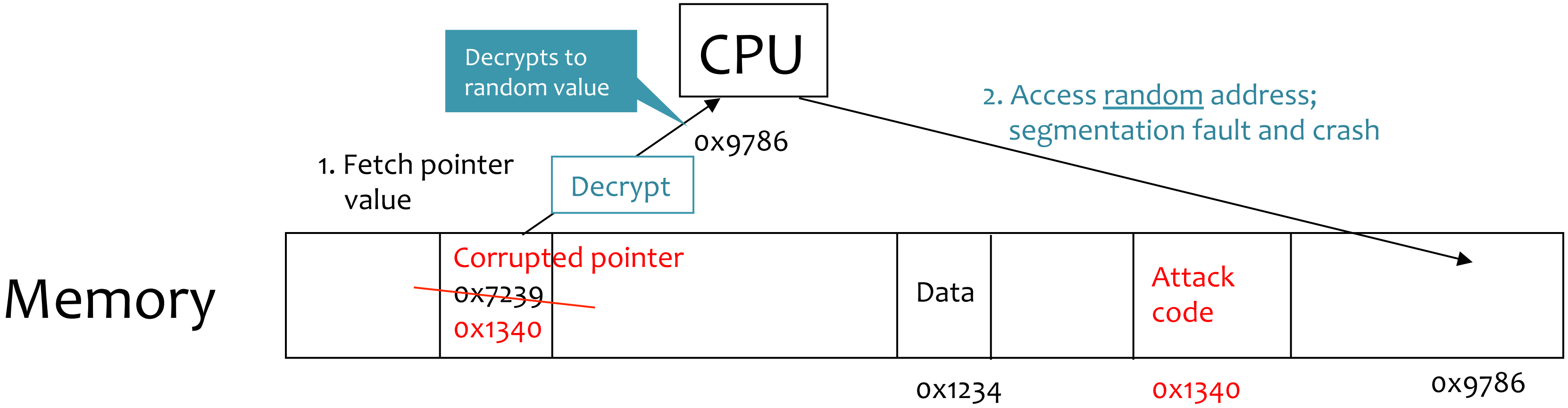
2. Access data referenced by pointer

Memory

Encrypted pointer
0x7239

Data

0x1234



Decrypts to random value

CPU

0x9786

2. Access random address; segmentation fault and crash

1. Fetch pointer value

Decrypt

Memory

Corrupted pointer
~~0x7239~~
0x1340

Data

Attack code

0x1234

0x1340

0x9786

PointGuard Issues

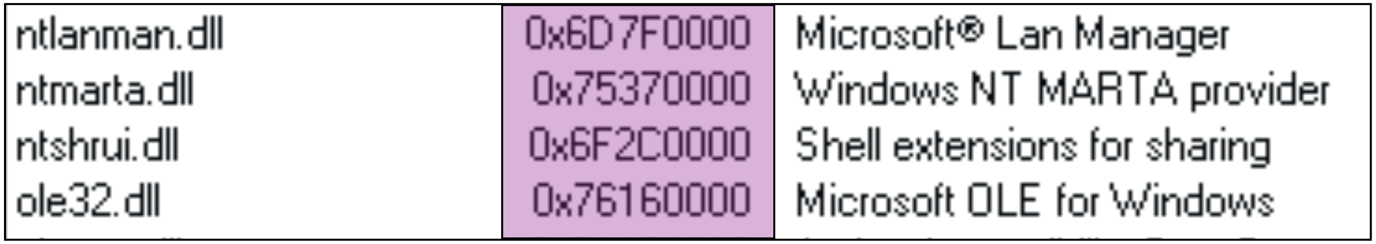
- Must be very fast
  - Pointer dereferences are very common
- Compiler issues
  - Must encrypt and decrypt <u>only</u> pointers
  - If compiler "spills" registers, unencrypted pointer values end up in memory and can be overwritten there
- Attacker should not be able to modify the key
  - Store key in its own non-writable memory page
- PG'd code doesn't mix well with normal code
  - What if PG'd code needs to pass a pointer to OS kernel?

ASLR: Address Space Randomization

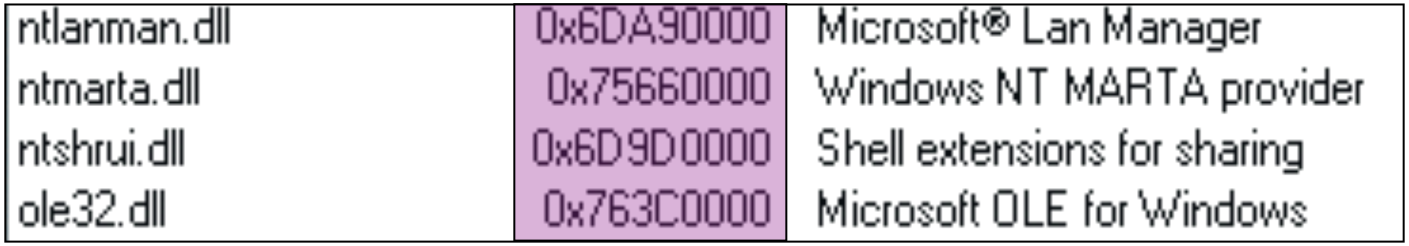
- Map shared libraries to a random location in process memory
  - Attacker does not know addresses of executable code
- Deployment
  - Windows Vista: 8 bits of randomness for DLLs
  - Linux (via PaX): 16 bits of randomness for libraries
  - More effective on 64-bit architectures
- Other randomization methods
  - Randomize system call ids or instruction set

Example: ASLR in Vista

- Booting Vista 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

ASLR Issues

- NOP slides and heap spraying to increase likelihood for custom code (e.g. on heap)
- Brute force attacks or memory disclosures to map out memory on the fly
  - Disclosing a single address can reveal the location of all code within a library

Other Possible Solutions

- Use safe programming languages, e.g., Java
  - What about legacy C code?
  - (Note that Java is not the complete solution)
- Static analysis of source code to find overflows
- LibSafe: dynamically loaded library that intercepts calls to unsafe C functions and checks that there's enough space before doing copies
  - Also doesn't prevent everything