ECE568 Lecture 03: Format String, Double Free and Buffer Overflow Defence

Wei Huang
Department of Electrical and Computer Engineering
University of Toronto

Lecture Outline

- Format String Vulnerabilities
  - Vulnerability itself
  - Exploiting Format String Vulnerabilities
- Double Free Vulnerabilities
  - Vulnerability itself
  - Exploitation
- Buffer Overflow Defence
  - …

Format String Vulnerabilities

1. Simple format string vulnerability:

```
sprintf(buf, “Some stuff %s”, attacker_string);
```

   - This is essentially the same as a *strcpy*, which results in a buffer overflow

2. More complex vulnerability

```
snprintf(buf, bufsize, attacker_string);
```

   - Here there is no buffer overflow as bufsize controls how many characters will be written into buf. However, the attacker gets to specify the format string.

Format String Functions

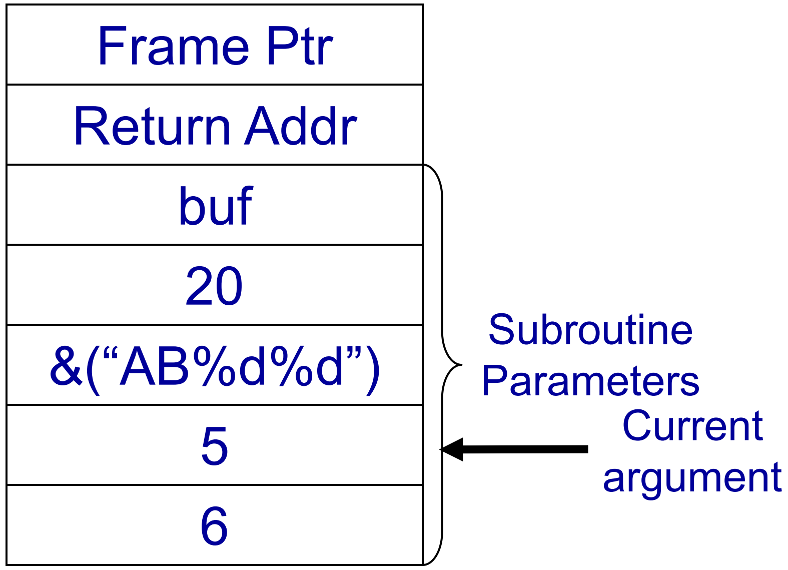
- Family of functions: printf, sprintf, snprintf, etc…
  - What does **printf (“%s”, val)** do?
  - What about **printf(“%d”, val)**?
  - What about **printf(“0x%x\n”, val)**

snprintf Operation

```
void main() {
  char buf[20];
  /* 20 specifies the max # of
   characters to print */
  snprintf(buf,20,"AB%d%d",5,6);
}
```

1. Arguments are pushed onto the stack in reverse order.
2. sprintf copies everything in the format string till it sees a "%".
3. A pointer to the current argument on the stack is then used to fill the next format character.
4. It is then moved to the next argument down the stack to get the next argument.

Contents of the Stack:



Contents of the buf:

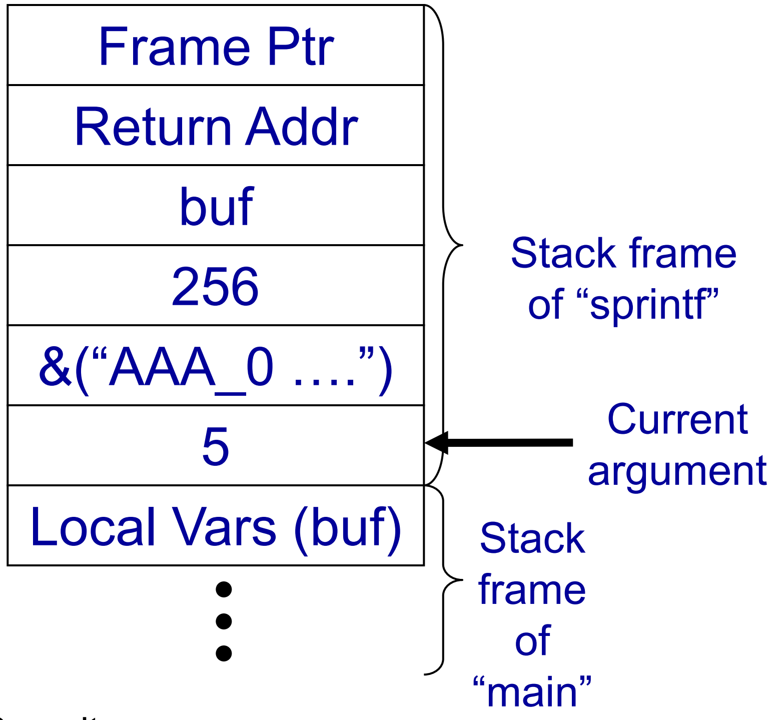
AB56

Unexpected Behavior

```
void main()
{
  char buf[256];
  snprintf(buf, 256, "AAA0_%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x\n",5);
  printf(buf);
}
```

- Now what happens if there are more "%" characters than arguments pushed on the stack?
  - The argument pointer keeps moving up the stack even though there are no arguments. It points to values in the previous stack frame

Contents of the Stack:



Unexpected Behavior

- The output of the program is:

```
AAA0_00000005.001bc4f8.00000000.bff63284.001ae639.008bea0e.30414141.303
    0305f.30303030
```

```
    Format String: ("AAA0_%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x\n")
```

- Note that 0x30414141 corresponds to our string "AAA0" (remember that the x86 is little endian). This means that the argument pointer has crawled its way into the next stack frame and is pointing into `buf` in main()!

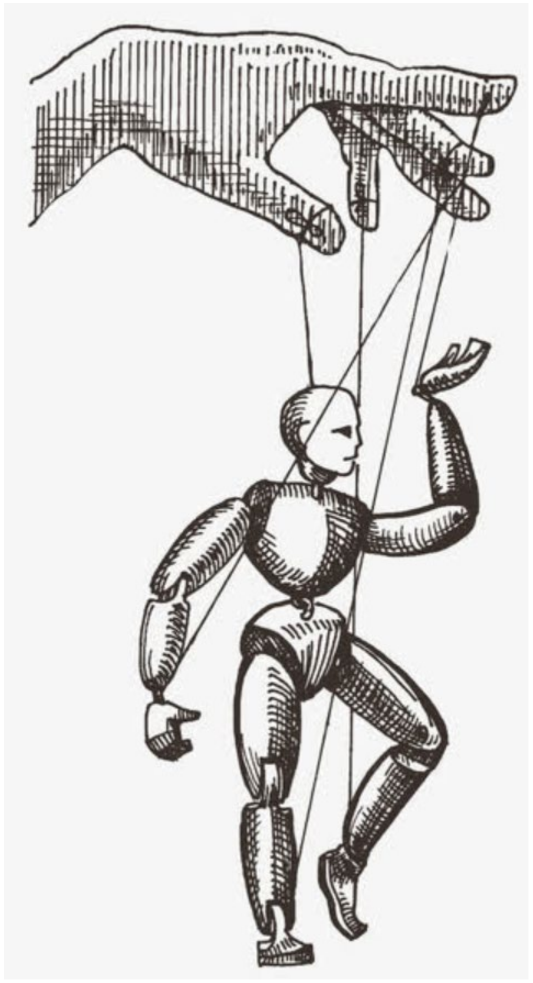
- But we have to somehow overwrite the return address, how do we do this?

Format String Function Man Page

```
PRINTF(3)                Linux Programmer’s Manual                PRINTF(3)

NAME
       printf,  fprintf,  sprintf,  snprintf,  vprintf,  vfprintf,  vsprintf,
       vsnprintf - formatted output conversion

SYNOPSIS
       #include <stdio.h>

       int printf(const char *format, ...);
       int fprintf(FILE *stream, const char *format, ...);
       int sprintf(char *str, const char *format, ...);
       int snprintf(char *str, size_t size, const char *format, ...);
…
The flag characters
          The character % is followed by zero or more of the following flags:


n         The number of characters written so far is stored into the
          integer  indicated  by  the int * (or variant) pointer argument.
          No argument is converted.
```

Arbitrary read and write primitives

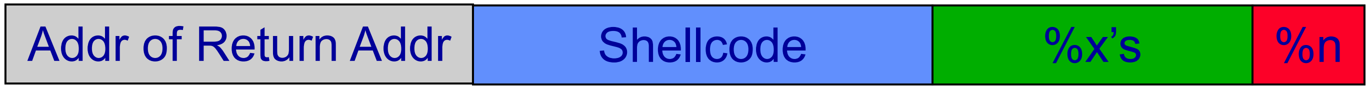
- This gives attackers an “arbitrary write primitive”
  - Attacker can control **what** is written by controlling the number of characters written
  - Attacker can control **where** data is written by controlling what the current argument pointer is pointing to when function hits the %n
- This almost gives attacker an “arbitrary read primitive”
  - Attacker can almost control **where** the current argument pointer points and can read from the location
  - Almost: argument pointer only goes **up** in memory, can’t read below the stack, but enough for the attack



Exploiting Format String Vulnerabilities

- Overall plan for format string attacks:
  1. Identify a format string function where you get to specify the format string.
  2. At the front of your format string, put the address where you think the return address is stored.
  3. Put your shellcode in the format string.
  4. Put enough "%" arguments so that the argument pointer points to the front of your format string
  5. Put a %n at the end and overwrite the return address to point at the shellcode in the buffer.

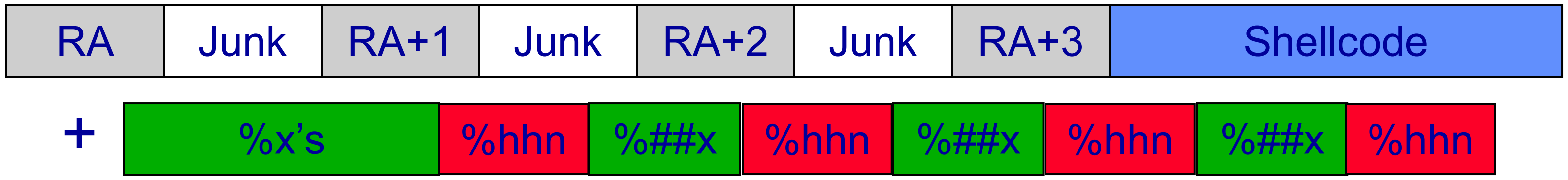
Attack Buffer:



Optimization

- Note that functions like snprintf will interpret the whole format string regardless of what the length of the output is. If the output is longer than the specified length, they truncate the output afterwards.

- You can control the number of characters written out by changing the number between the "%" and x,u or d. For example %243u will write out 243 characters exactly.
  - However, creating a 32-bit number to overwrite the return address would mean that you'd have to write out A LOT of characters (more than will fit in memory actually). Fortunately, what you can do is write the 32-bit number 1 byte at a time.



- For more info, see the article:
  - "Exploiting Format String Vulnerabilities" by Scut, posted on the website and in Lab1 tarball

Example

- The attacker wants to write 0x10121110 to return address at 0xbffffa0, there are **4 bytes** between **format string** and **buffer**, with **no arguments**. How to build attacking string?

Double Free Vulnerability

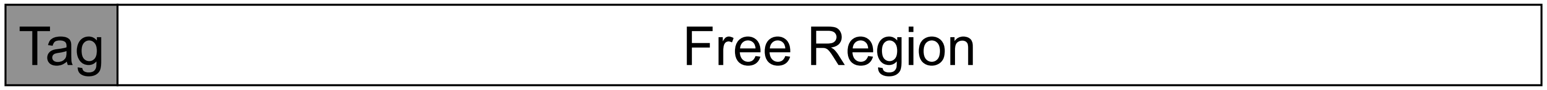
- Freeing a memory location that is under the control of an attacker is an exploitable vulnerability. For example:

```
p = malloc(128);
q = malloc(128);
…
free(p);
free(q);
p = malloc(256);
strcpy(p, attacker_string);
free(q);  ←——— Vulnerability!
```

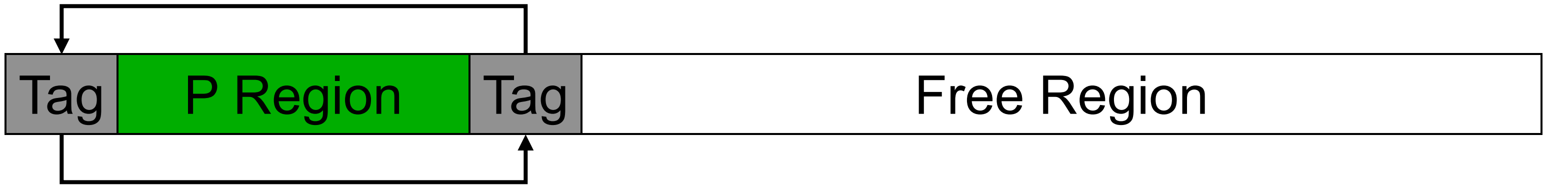
- Why is this vulnerable?
  – To understand, lets look at how malloc works

Malloc

- Malloc maintains a doubly link-list of free and allocated memory regions:
  - Information about regions is maintained in *Chunk Tag* that is stored just before a region. Each chunk maintains:
    - Whether the chunk is allocated or free in the *free bit*
    - Links to the next and previous chunk tags
  - Initially when all memory is unallocated, it is in one free memory region

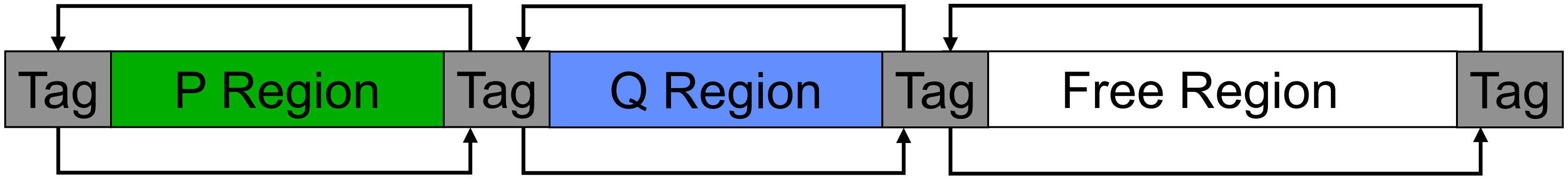


  - When a region is allocated, malloc creates two regions, one is free, and one is allocated:

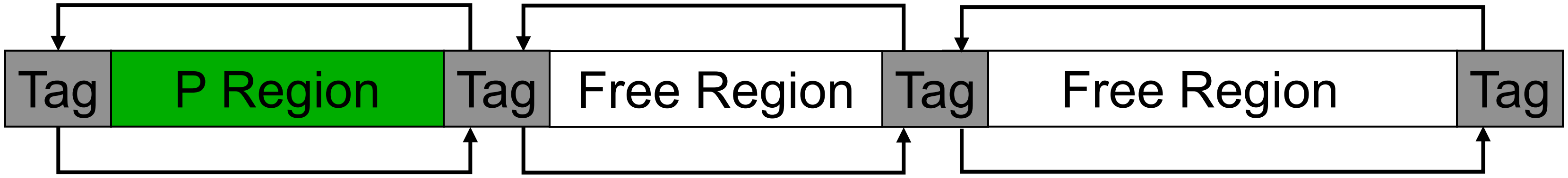


Malloc

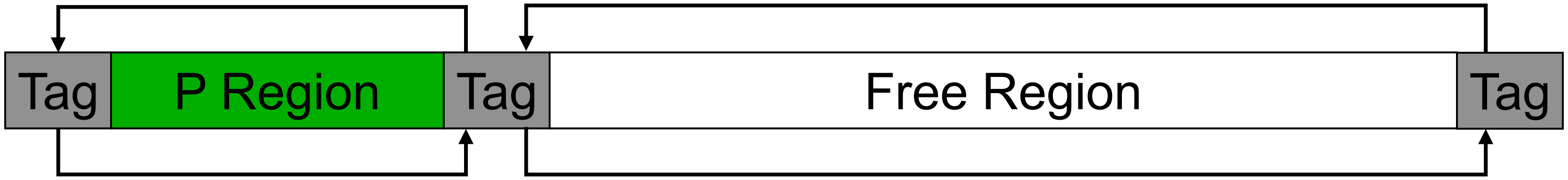
- When another region is allocated, another tag is created:



- When regions are deleted, the free() function sets the free bit



- Free() also tries to consolidate adjacent free regions



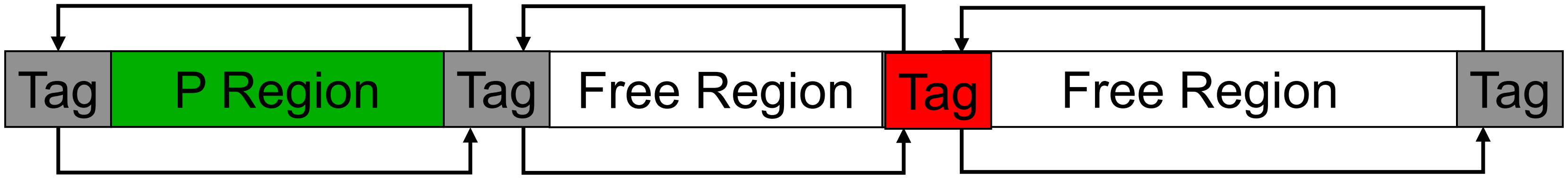
The Edward S. Rogers Sr. Department of Electrical & Computer Engineering UNIVERSITY OF TORONTO

Code for free()

- To consolidate:
  - free() removes a tag element from the list

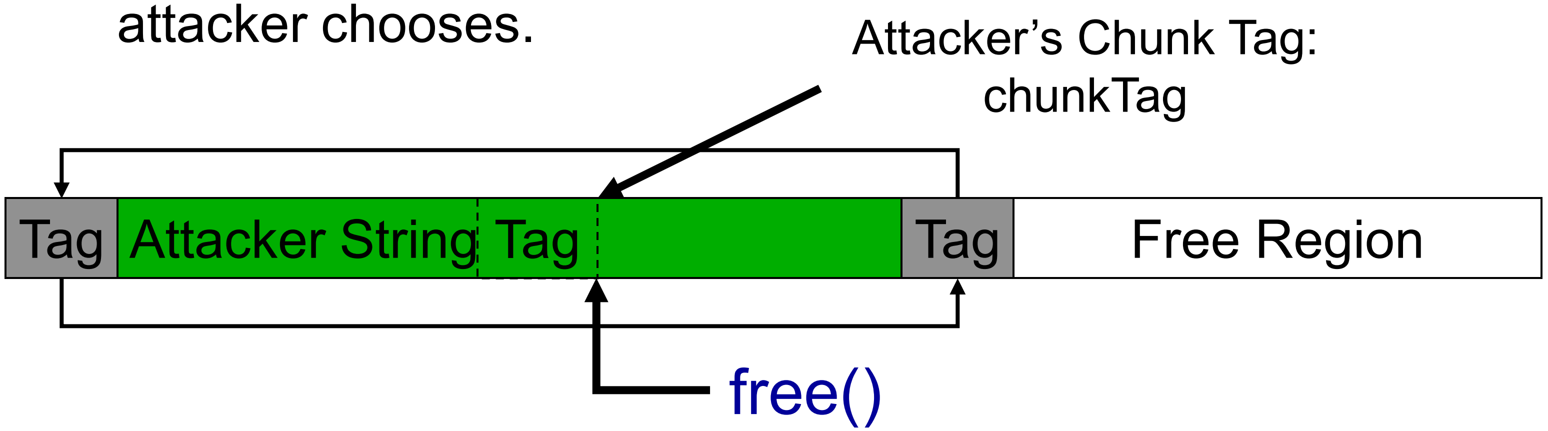
- Code:

```
tag->prev->next  = tag->next
Tag->next->prev  = tag->prev
```

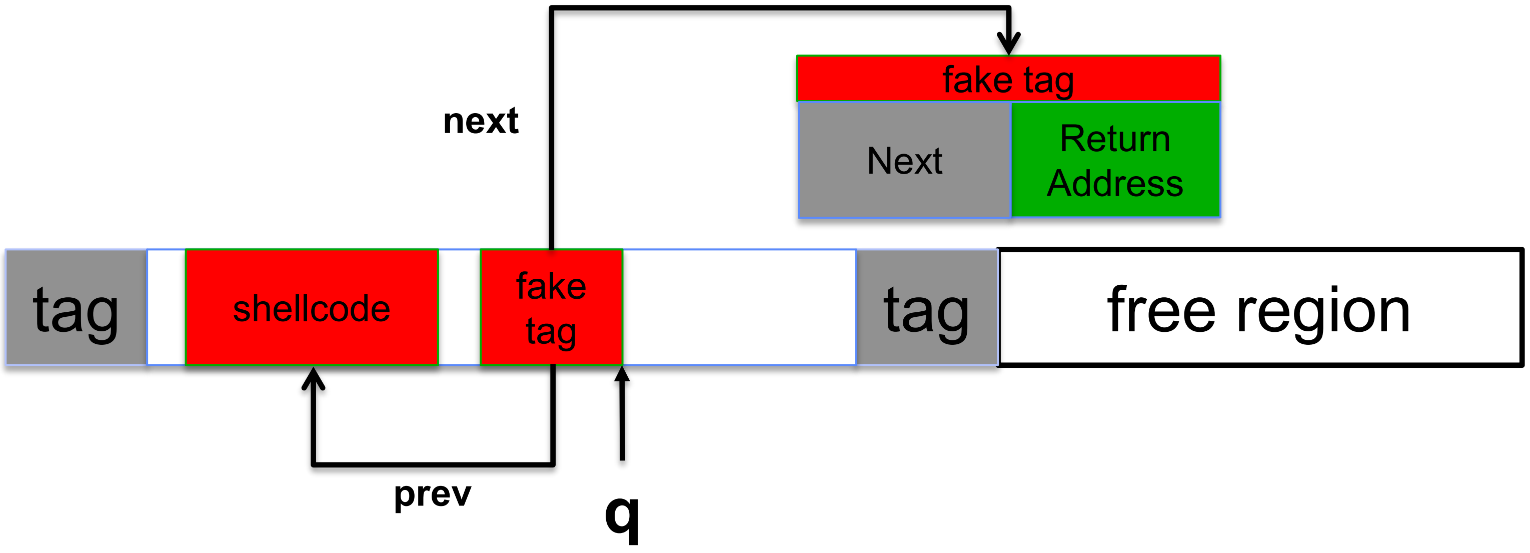


Double-Free Vulnerability

- A vulnerability occurs when the program calls free on a region that contains data set by the attacker:
  - The free() function will look at the address just before the address where the program calls free to find the chunk tag
  - In this case, the chunk tag is part of the attacker's string
  - The attacker can set the value of the chunk tag such that she can make free() overwrite a location in memory of the attacker's choice (like a return address) with a value that the attacker chooses.



Double-Free Vulnerability



When consolidating free regions, **free** essentially does:

```
tag = q - sizeof(chunkTag);
tag->next->prev = tag->prev;
```

Break

- Be back in 10 minutes
- Next session: Defence to Buffer Overflow Attacks

Defence to Buffer Overflow Attacks

- Preventing return address overwrite
- Preventing execution of injected code
- Making it hard to guess the target address

Defenses

- The most obvious way to defend against buffer overflow and format string vulnerabilities is not to make them:
  - Rigorous code audits
  - Using a type safe language that does bounds checking (i.e. JAVA, ADA)
    - This means code will be memory safe (i.e. compiler will enforce the memory access rules of the language)

- However, this is not always possible:
  - Too much code already exists to move to another language
  - Source code is not available
  - Performance may be a concern

Desired defense properties

Requires no changes to source code
- Many programs are large
- Vulnerabilities will be present, so must prevent exploitation

Low overhead:
- Defense does not make program execute slowly (performance)
- Defense does not use up too much extra memory

Other desirables:
- Only takes effect if an attack actually occurs (accuracy)
- Does not require source code or specialized knowledge to use (usability)

A Generic Defense?

Buffer overflow attack requires an input string to be copied into a buffer without bounds checking

- Typical attack requires three steps
  1. Overwrite code pointer such as return address
  2. Inject shell code
  3. Redirect execution to shell code

What is needed for these steps to succeed?

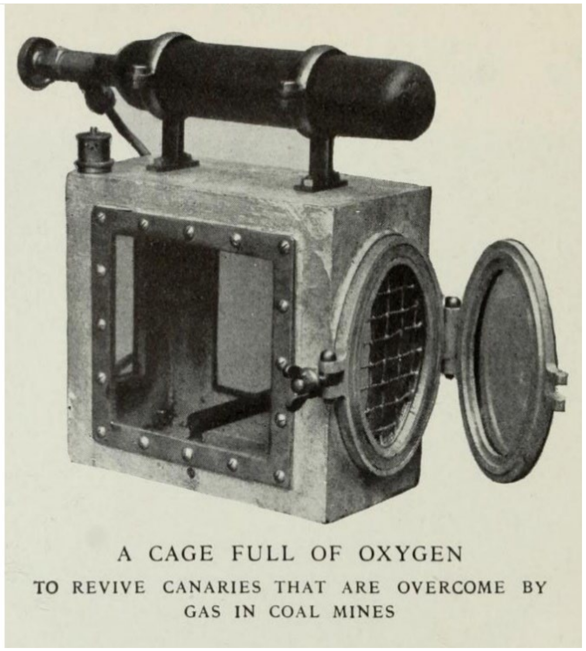
1. Ability to overwrite return address
2. Injected code has to be executable
3. Target address has to be guessed

Let's look at how to detect or prevent each of these steps…

Defense #1: Preventing overwrite of return address

Stack Canary

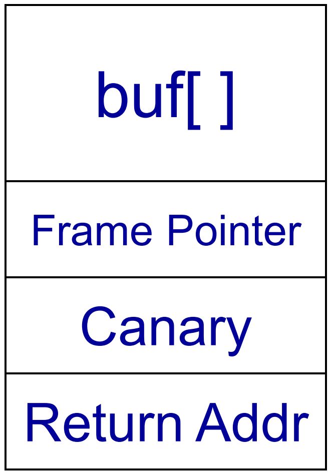
- Miner's canary



Defending Against Stack Smashing

- Recent protection techniques will prevent the return address from being overwritten:
  - Stackshield: Put return addresses on a separate stack with no data
  - Stackguard: Crispin Cowan's solution [previous lecture's ref]

- How Stackguard works:
  - On a function call, a random "Canary" Value is placed just before the return address.
  - When the function returns, it first checks the canary value. If the value has changed, program execution is halted.
  - Support in both GCC and MSVC++



Response: Function Pointers

- Rather than overwriting the return address, the adversary can try to overwrite a function pointer
- Recall:
  - A function pointer is a variable that can be dereferenced to call a function:

```
int foo(int arg1) {
...
}
/* define a function pointer */
int (*fp)(int arg1);
/* assign the address of a function to the pointer */
fp = &foo;
/* call the function via the pointer */
fp(6);
```

Function Pointers

- Function pointers are often used in C to mimic polymorphism that's supported in object oriented languages. The are also useful to enable changing functions at run time:
  - Very common in OS kernels where the kernel has to run with different modules or drivers without recompilation.
  - Also common in other programs that use modules such as web servers, etc…
  - Used to support dynamically loaded libraries.
  - Common in object-oriented languages (i.e. C++ vtables)

- Sometimes the buffer will not be close enough to the return address, but will sit next to a function pointer.
  - By overwriting the function pointer, the attacker can cause execution to be redirected next time the program calls the function pointer.

Dynamic Linking

Program code needs to call functions such as `printf` in dynamic libraries

- These libraries are normally linked into the program at run time, at arbitrary locations, by a dynamic linker
- Typically, both the caller of a library function and the function itself are compiled to be **position independent**
- We need to map the position independent function call to the absolute location of the function’s code in the library
  - The dynamic linker performs this mapping
  - It uses two tables: the **Procedure Linkage Table** (PLT) and the **Global Offset Table** (GOT)

PLT/GOT

GOT is a table of function pointers:

- Contains the absolute memory location of each of the dynamically-loaded library functions
- Locations are only known at runtime

PLT is a table of small functions:

- One function in table per library function used by the program.
- First time it’s called, the PLT function invokes dynamic linker to fill in the location of the function in the GOT.
- Also load library from disk if the first time any code from the library is called

PLT/GOT Overwrites

Suppose that an attacker is only able to overwrite a single chosen address location with a chosen value

- Then a good option is to overwrite a GOT function pointer

A binary utility like **`objdump -x`** allows dissassembling an executable

- It provides the location of these structures
- PLT/GOT always appear at a **known** location

Note this is specific only to ELF binaries used in UNIX systems. Windows and Mac systems use slightly different method.

Response: Argument Overwrite

- If she can't redirect execution, she may also be able to affect an argument passed to an *exec* system call. For example, a program has:

```
char buf[128] = “my_program”;
char vulnerable[32];
exec(buf);
```

  - The attacker can **corrupt the argument** `buf` by overflowing vulnerable and have the program execute something else.

Response: Bad Bounds Check

Who can see what's wrong with this bounds check?

```
/* Linux 2.4.5/drivers/char/drm/i810_dma.c */
if(copy_from_user(&d, arg, sizeof(arg)))
       return -EFAULT;
if(d.idx > dma->buf_count)
       return -EINVAL;
buf = dma->buflist[d.idx];
copy_from_user(buf->virtual, d.address, d.used);
```

- While this vulnerability doesn't allow the attacker to hijack the program, such vulnerabilities have led to remote code execution in the past (e.g. the *do_brk()* function in the Linux 2.4.22 kernel). They can also let the attacker overwrite arbitrary memory locations.

Defense #2: Preventing execution of injected code

Defense: Non-Executable Pages

- As of 2004 all Intel and AMD processors support non-executable pages (pages can only contain data, not code).
  - This means the shellcode in your buffer won't execute since it is only interpreted by the processor as data. Called "Data Execute Prevention" (DEP) on Windows systems
  - However, the attacker can still redirect execution, so she redirects it to a function that already exists in the code that accomplishes the same thing. This is called a **return-into-libc** attack.
  - For example, she may find another place in the program that calls *exec* or *system* which can also be used to execute an arbitrary program.

Response: Return into LibC

- Another way to avoid injecting new code is to use code that is already present:
  - Many libc functions have code that is useful to the attacker. For example, the `system` call (from man page):

```
int system(const char *string);
•  system()  executes a command specified in string by
   calling /bin/sh -c string,
```

Isn’t this just as good as your own shell code? How can attacker use this?

Return-oriented programming (ROP)

What if the desired attack code can't be performed with only one libc function?

- Can string together libc calls with a corrupted stack
- When first function returns, it looks on the stack to find the return address to the next function

Observation:

- You can inject multiple calls in a corrupted stack

Return-oriented programming

Method:

- Note that there is a lot of code in standard libraries (i.e. libc)
- You don't have to return to the start a function, you can return anywhere
  - You can find return instructions in the middle of functions
  - Because x86 instructions are variable, you can actually return into the middle of an instruction!
- Because of this, you can find "gadgets" – small sections of code that execute any instruction you want, followed by a return
  - Turing complete! (read ROP paper in course ref)
- Drawbacks for the attacker:
  - Large attack buffers required
  - Lots of returns generated

Example

- Suppose adversary has the following gadgets available. How can she construct a stack frame that will execute the system call exit(-1)?

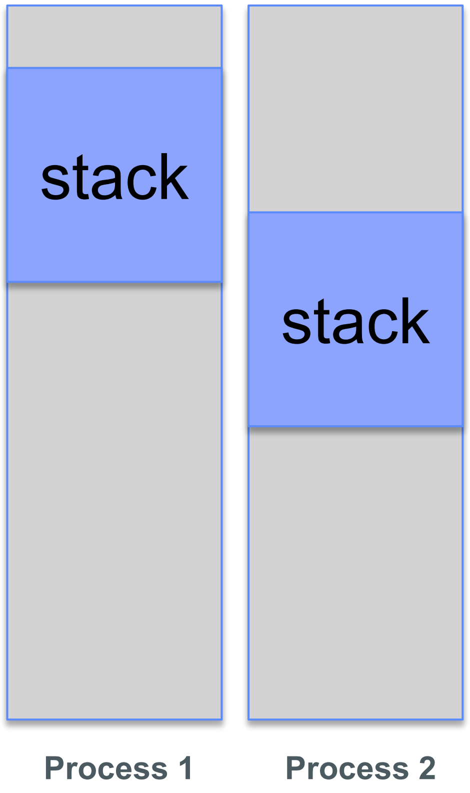
```
0x00a12345:       int  0x80
                  ret
0x00a19425:       mov 0x0, eax
                  ret
0x00a29493:       mov 0x1, ebx
                  ret
0x00a31495:       add ebx, ebx
                  ret
0x00a35946:       pop ebx
                  ret
0x00a36723:       push ebx
                  ret
```

Defense #3: Preventing guessing the target address

Defense: Address-Space Layout Randomization (ASLR)

Recall that the target address (*e.g.*, the buffer's location on the stack) has to be guessed:

- With ASLR, the OS maps the stack of each process at a **randomly** selected location with each invocation
  - An attacker will not be able to easily guess the target address
  - Application will crash rather than executing the attacker's code
  - ASLR also randomizes location of dynamically loaded libraries, making it harder to perform return-into-libc attacks or GOT overwrites
- Linux 2.6, Android 4.0 and Windows Vista and later use ASLR



Guessing address

A lot of server code will restart crashed processes to maintain availability

- This gives the attacker a chance to attack the same application repeatedly and guess different locations.
- Computers are very fast, can do many guesses quickly
- To work, ASLR must have enough entropy to randomize from
  - Need large address space to have many possibilities
  - There are restrictions like stack being at the top of memory, code at the bottom
  - ASLR is much more effective with 64-bit code than 32-bit code.

Address space leakage

Format string-like vulnerabilities can leak address space contents or pointers. Requires 2 step attack:

- Step 1: Learn the location of a known object, enabling attacker to know the layout
- Step 2: Exploit vulnerability and overwrite code pointer with known location

Heap spray

If vulnerability allows overwrite of large sections of heap, just write many copies of the shellcode and hope that we can jump into one:

- Common in dynamic compilation environments where NX-pages aren’t use
- JVMs, javascript engines, etc…

Control-flow Integrity

Might be the **ultimate defense** against memory corruption attacks.

- All possible execution paths through a program are extracted from the source code
- Checks are added to the compiled binary to enforce that only paths explicit in the source code are taken.

Example

```
void foo(char *s) {
  char buf[32];
  strcpy(s,buf);
  return;
}

int main(int argc, char *argv[]) {
  void (*f)(char*) = &foo;
  f(argv[1]);
  system("echo \"hello world\"");
}
```

Control flow integrity

Properties:

- No changes to source code (though source code is required)
- Can have **significant** performance overhead. In particular:
  - Returns from commonly called functions can return to many call sites. This requires checking return against a long list of possible values.
  - An alternative is to maintain a separate stack for return addresses and prevent it from being overwritten. This is expensive as it requires checking every memory access to make sure it doesn't modify the protected stack.

ROP Defenses

- Lower overhead defenses:
  - Processors maintain a “Last Branch Record” (LBR), which is the source and destination of the most recent returns executed in hardware
  - Periodically check the LBR for anomalies. I.e. too many returns for number of instructions executed. Can also check just before sensitive system call (i.e. exec, system …)
- Can have very low overhead ~5%

ROP Defenses

- kBouncer:
  - Proposal from grad students at Columbia
  - Won ~200K at Microsoft Bluehat contest http://www.microsoft.com/security/bluehatprize
- ROPecker:
  - More advanced version of kBouncer
- Unfortunately, broken again:
  - See paper “ROP is Still Dangerous: Breaking Modern Defenses”

Intel MPX/CET and ARM PAC

- As of September 2015, Intel has added *Memory Protection Extensions (MPX)* to 6th Generation Intel processors (Skylake)
- Provide 4 additional registers that hold upper and lower bounds for a buffer (Registers `BND0..3`)
  - Provide 2 upper and lower bound check instructions (`BNDCL` and `BNDCU`)
  - Allow hardware supported bounds checking. If bound check fails, exception is thrown
- Likely subsumed by Intel CET support:
  - Hardware support for hardware checking of branch targets and return addresses
  - Available in the latest Tiger Lake Gen 11 processors, General availability: Nov 2020

Intel MPX/CET and ARM PAC

- ARM Pointer Authentication (PAC)
  - Observe that even though pointers are 64-bits, few machines have enough memory to use all 64-bits. Typically only ~40 bits are used
  - Use the top ~20bits as a cryptographic checksum (we'll be talking about MAC's in a couple of weeks)
  - If pointer is overwritten, adversary does not know the correct checksum to write, so tampering will be detected by hardware
  - Main concern is ~20-bits is not a lot of possibilities, adversary may be able to guess the correct value with enough tries.

Vulnerability Databases

- To aid computer administrators, there are several large databases of vulnerabilities on the Internet:
  - National Vulnerability Database (http://nvd.nist.gov)
  - CERT (https://www.kb.cert.org/vuls/)
  - VulDB (https://vuldb.com)
  - MITRE CVE (https://www.cve.org)
  - Exploit Database (https://www.exploit-db.com)
- For any program and version, one can do a lookup from these databases and get a description of the vulnerability.

Conclusion

- Easy to make a mistake and end up with a vulnerability.
- Exploiting them takes a bit of work, but is not beyond someone who knows what they are doing.
- Certain vulnerabilities can be removed by moving to safer languages:
  - A lot of vulnerabilities result from uses of pointers and running off the end of arrays.
  - Java doesn't allow the use of pointers, does array bounds checking automatically and has a stronger type system.
- However, the only real defense is to be aware of what vulnerabilities exist, to be extra careful creating code and let others audit your code.