EngSci Year 3 Winter 2022 Notes

Brian Chen

Division of Engineering Science
University of Toronto
https://chenbrian.ca

brianchen.chen@mail.utoronto.ca

Contents

Section 1

CSC473: Advanced Algorithms

Subsection 1.1

Global Min-Cut (Karger's Contraction Algorithm)

Given an undirected, unweighted, and connected graph $G = (V, E)$, **return** the smallest set of edges that disconnects G

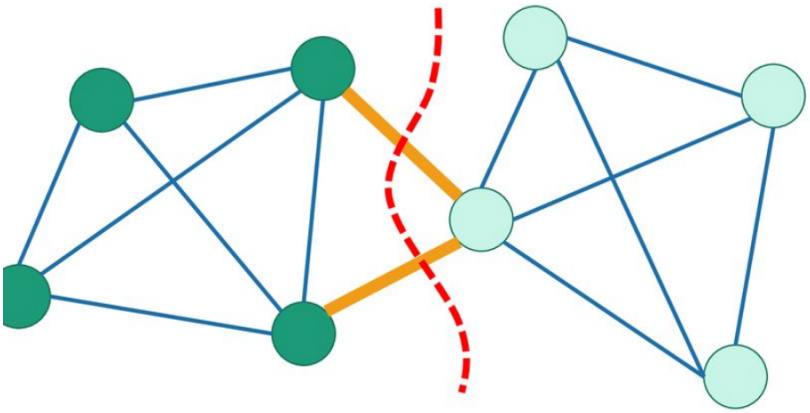


Figure 1. Example of global min-cut. Note that the global min-cut is not necessarily unique

Lemma 1 | If the min cut is of size $\geq k$, then G is k-edge-connected

An example of where this may be useful is in computer networks where we can measure the resiliency of a network by how many cuts must be made before a vertex (or many) get disconnected

It may be more convenient to return a set of vertices instead

Definition 1

$$S, T \subseteq V, S \cap T = \varnothing \tag{1.1}$$

$$E(S, T) = \{(u, v) \in E : u \in S, v \in T\} \tag{1.2}$$

The global min-cut is to output $S \subseteq V$ such that $S \neq \varnothing, S \neq V$, such that $E(S, V \setminus S)$ is minimized.

Comment

Note that the min-cut-max-flow problem is somewhat of a dual to the global min-cut problem; the min-cut-max-flow problem imposes a few more constraints than the global min-cut algorithm i.e. having a directed and weighted graph as well as the notion of a source or sink.

- **Input:** Directed, weighted, and connected $G = (V, E)$, $s \in V, t \in V$
- **Output** : S such that $s \in S, t \notin S$ such that $|E(S, V \setminus S)|$ is minimized

We can kind of intuitively see that the global min-cut can be taken to the minimum of all max-flows across the graph. So we can take the max-flow solution and then reduce it to find the global min cut.

Question: how many times will we have to run max-flow to solve the global min-cut problem? Naively, we may fix t to be an arbitrary node, then try every other $s \neq t$ to find the $s - t$ min-cut to get the best global min-cut.

We know from previous courses that the Edmonds-Karp max-flow algorithm will run in $O(nm^2) = O(n^5)$, which makes our global min-cut algorithm $O(n^6)$. However, there is a paper recently published which gives an algorithm for min-cut in nearly linear time, i.e $O(m^{1-O(1)}) = O(n^2)$ which gives a global min-cut runtime of $O(n^3)$.

A randomized algorithm will be presented that solves this problem in $O(n^2 \log^2 n)$

Definition 2 The **Contraction** operation takes an edge $e = (u, v)$ and *contracts* it into a new node w such that all edges connected to u, v now connect to w and u, v are removed. Note that the contracted nodes can be supernodes themselves.

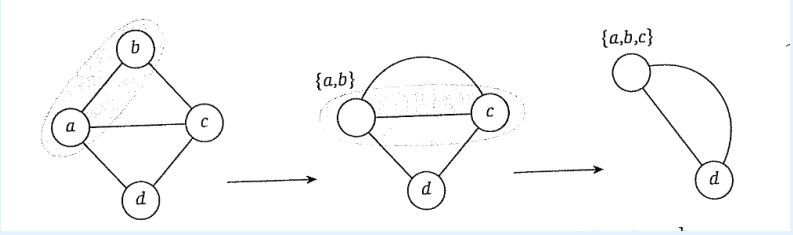


Figure 2. Example of a series of contractions

CONTRACTION$(G = (V, E))$

1 **while** G has more than 2 supernodes
2 Pick an edge $e = (u, v)$ uniformly at random
3 Contract e, remove self-loops
4 Output the cut $(S, V \setminus S)$ corresponding to the two super nodes

The contraction algorithm then recurses on G', choosing an edge uniformly at random and then contracting it. The algorithm terminates when it reaches *a* G' with only two supernodes v_1, v_2. The sets of nodes contracted to form each supernode $S(v_1), S(v_2)$ form a partition of V and are the cut found by the algorithm.

1.1.1 Analysis

The algorithm is still random, so there's a chance that it won't find the real global min-cut. With some analysis we will show that the success polynomial is not exponential as one may think, but really only polynomially small. Therefore by running the algorithm a polynomial number of times and returning the best cut identified we can find a global min-cut with high probability.

Lemma 2 The contraction algorithm returns a global min cut with probability at least $\frac{1}{\binom{n}{2}}$

Proof Take a global min-cut (A, B) of G and suppose it has size k, i.e. there is a set F of k edges with one end in A and the other in B. If an edge in F gets contracted then a node of A and a node in B would get contracted together and then the algorithm would no longer output (A, B), a global min-cut. An upper bound on the probability that an edge in F is contracted is the ratio of k to the size of E. A lower bound on the size of E can be imposed by noting that if any node v has degree $< k$ then $(v, V \setminus v)$ would form a cut of size less than k – which contradicts our first assumption that (A, B) is a global min-cut. So the probability than an edge in F is contracted at any step is

$$\frac{k}{\binom{k}{2}} = \frac{2}{n} \tag{1.3}$$

Note that here we use the number of vertices instead of the number of edges since each contraction removes (combines) one vertex, whereas since the amount of edges after a contraction can be very difficult to calculate.
Next, let's inspect the algorithm after j iterations. There will be $n - j$ supernodes in G' and we can take that no edge in F has been contracted yet. Every cut of G' is a cut of G, so there are at least k edges incident to every supernode of G'[1]. Therefore G' has at least $\frac{1}{2}k(n-j)$ edges, and so the probability than an edge of F is contracted in $j+1$ is at most

[1] *since the min-cut has k edges*

In terms of edges, it would be $\frac{k}{m_{i-1}}$, but again, edges are difficult to work with so we'll do it w.r.t the vertices/supernodes

$$\frac{k}{\frac{1}{2}k(n-j)} = \frac{2}{n-j} \tag{1.4}$$

$$P(A_i|A_1 \dots_{i-1}) \geq \frac{n-i-1}{n-i+1} = 1 - \frac{2}{n-i+1} \tag{1.5}$$

The global min-cut will be actually returned by the algorithm if no edge of F is contracted in iterations $1 - n$.
What we want to know, then, is what is the probability of this algorithm never making a mistake?

This prof uses commas to indicate intersection...

$$P(A_1 \dots A_{n-1}) = P(A_1)P(A_2|A_1)P(A_3|A_1, A_2) \dots P(A_{n-2}|A_1, A_2, \dots, A_{n-3}) \tag{1.6}$$

From what we found previously we know that this is

$$\geq \frac{n-2}{n} \times \frac{n-3}{n-1} \times \frac{n-4}{n-2} \dots \times \frac{2}{4} \times \frac{1}{3} = \frac{2}{n(n-1)} = \frac{1}{\binom{n}{2}} \tag{1.7}$$

□

This gives us a bound of $O(n^2)$ using the n^2 term from the number of contractions and then n^2 to get correct output with constant probability of success.

The key observation is that early contractions are much less likely to lead to a mistake, which leads us to the Karger-Stein min cut.

Subsection 1.2

Karger-Stein Min Cut Algorithm

This algorithm solves the global min-cut problem in $O(n^2 \log^2 n)$ by taking advantage of the earlier cuts; it stops the contraction algorithm after an arbitrary fraction of contractions steps and then recursively contracts *more carefully*.

Exercise: show the following:

The probability of no mistake in the first i contractions

$$P(A_1, A_2, \dots A_i) \geq \frac{(n-i)(n-i-1)}{n(n-1)} \tag{1.8}$$

MIN-CUT$(G = (V, E))$

1 **if** G has two supernodes corresponding to $S, \hat{S}$
2 **return** $S, \hat{S}$
3 Run the contraction algorithm until $\frac{n}{\sqrt{2}} + 1$ supernodes remain
4 Let G' be the resulting contracted multigraph
5 $(S_1, \hat{S}_1) =$ PIN-CUT(G')
6 $(S_2, \hat{S}_2) =$ PIN-CUT(G')
7 **return** the cut $(S_i, \hat{S}_i)$ with the smaller number of edges

Theorem 1 MIN-CUT(G) runs in $O(n^2) \log n$ and outputs a min cut of G with probability of at least $\frac{1}{O(\log n)}$[2]

[2] *So repeat the algorithm $O(\log(n))$ times, leading to $O(n^2 \log^2 n)$ runtime*

Comment

Note that there are two types of randomized algorithms:

- Monte carlo algorithms: bound on worst-case time & produces a correct answer with a probability $\geq$ some constant
- Las vegas algorithms: bound on the expected value of running time, but the output is always correct

Our contraction algorithm is a monte-carlo algorithm

Section 2

ECE568 Computer Security

Subsection 2.1

Refresher Introduction

Software systems are ubiquitous and critical. Therefore it is important to learn how to protect against malicious actors. This course covers attack vectors and ways to design software securely

Data representation: It's important to recognize that data is just a collection of bits and it is up to us to tell the computer how it should be interpreted. Oftentimes we can make assumptions, for example assume that an int is an int. But what if we end up being wrong about it? Many security exploits rely on data being interpreted in a different way than originally intended. For example,

```
1 unsigned long int h = 0x6f6c6c6548; // ascii for hello
2 unsigned long int w = 431316168567; // ascii for world
3 printf("%s %s", (char*) h, (char*) w);
```

Listing 1: An innocent example of where we should be careful about data representation. This prints hello world

This courses makes use of Intel assembler. TLDR:

- 6 General-purpose registers
- RAX (64b), EAX(32b), AX(16b), AH/AL(8b), etc

Stack grows

Subsection 2.2
Software Code Vulnerabilities

Recall: the stack is used to keep track of return addresses across function calls; storing a breadcrumb trail. Another key thing sitting in the stack are local variables. A common theme in the course is that computing tends to conflate execution instructions with data.

Common data formats and structures create an opportunity for things to get confused (and for attackers to take advantage of). For example, a buffer-overflow attack can end up overwriting that return address breadcrumb trail and then execute arbitrary code.

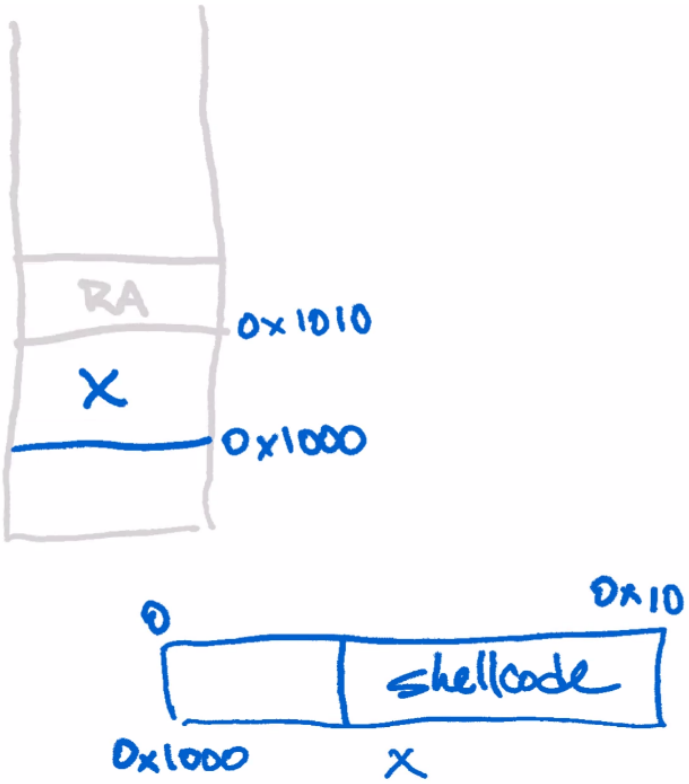


Figure 3. Bufferoverflow to write to the return address. Shellcode is a sequence of instructions that is used as the payload of an attack. It si called a shellcode because they commonly are used to start a shell from which the attacker can do more.

There are ways to find out where that return address is (or at least reasonably guess). This is discussed more in detail later; for now we'll assume that they have it figured out.

A common technique to make this easier is to inject a bunch of `NOP`s before the start of the shellcode. So that we don't need to be as precise as needed in order to find the shellcode start.

One technique for finding the RA would be to incrementally increase the size of the buffer overflow until we get a segfault – at this point the segfault would tell you what memory address it was trying to access and possibly the values it saw there instead as well.

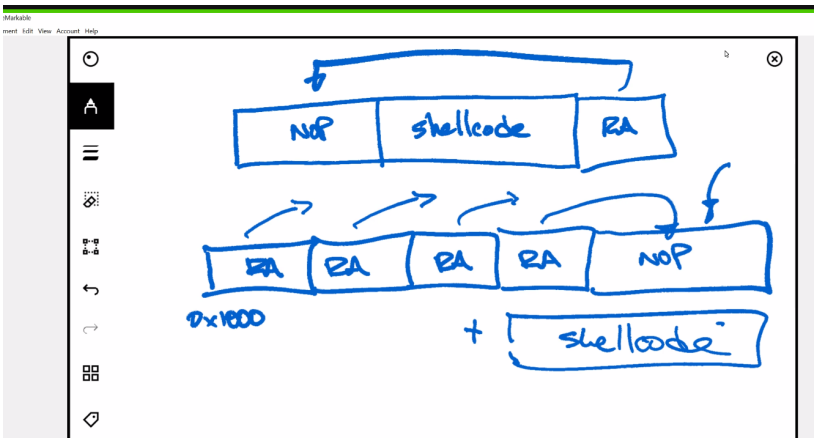


Figure 4. Can create a RA sled with a NOP leading to shellcode and then try it from e.g. 0x1000, 0x2000 and so forth to find where to attack from.

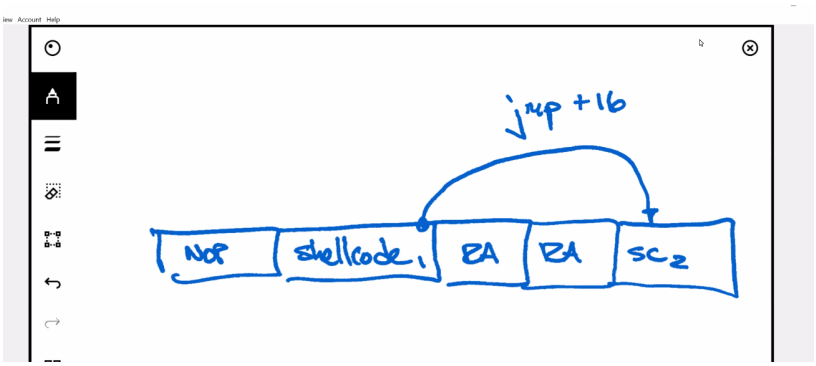


Figure 5. Another technique may involve placing shellcode all over the place, of which each one may be a valid entrypoint into the shellcode.

Subsection 2.3

Format string Vulnerabilities

```
1 spirntf(buf, "Hello %s", name);
```

`sprint` is similar to `printf` except the output is copied into `buf`. The vulnerability is simiar to the buffer overflow vulnerability. The difference is that the attacker can control the format string.

Consider the following:

```
1 char* str = "Hello world";
2 printf(str); // 1
3 printf("%s", str); // 2
```

Despite it looking different there are differences in these two ways to print hello world. The first argument is a format string, which is different from just a parameter. A format string contains both instructions for the `c` printing library as well as data. This means that the first method can be exploited if the attacker has access to the format string. A more complex vulnerability is with `snprintf` (which limits the number of characters written into buf).

```c
void main() {
    const int len = 10;
    char buf[len];
    snprintf(buf, len, "AB%d%d", 5, 6);
    // buf is now "AB56"
}
```

- Arguments are pushed to the stack in reverse order
- snprintf copies data from the format string until it reaches a %. The next argument is then fetched and outputted in the requested format
- What happens if there are more % parameters than arguments? The argument pointer keeps moving up the stack and then points to values in the previous frame (and could actually look at your entire program memory, really)

```c
void main () {
    char buf[256];
    snprintf(buf, 256,
    "AB,%08x,%08x,%08x,%08x,%08x,%08x,%08x,%08x,%08x,%08x", 5);
    printf(buf);
    //
    AB,00000005,00000000,29ee6890,302c4241,2c353030,30303030,39383665,32346332,33353363,30333033%
    // if we look at the 3rd clause as ascii we get '0,BA' (recall
    intel little endian) i.e. we've read up far enough to see the
    local variable specifying the format string pushed onto the stack
    earlier
}
```

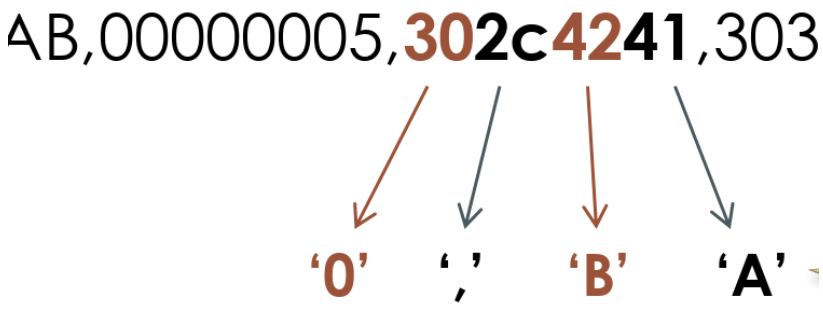
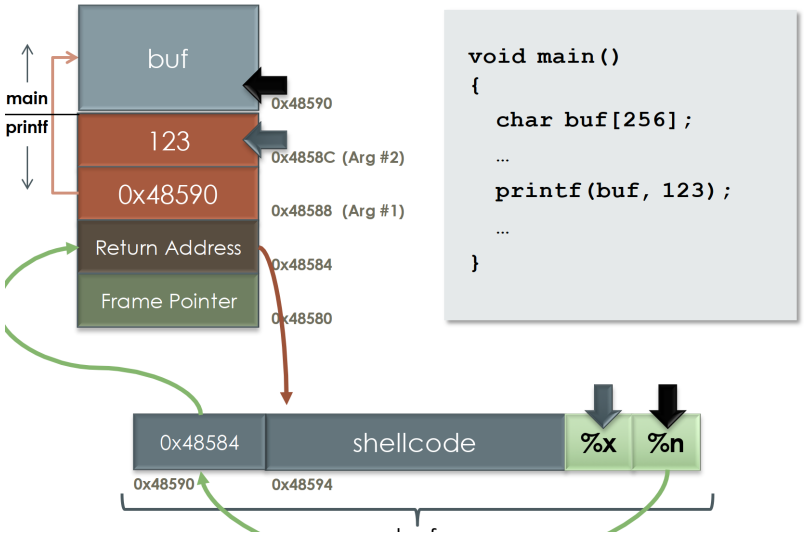


Figure 6. ASCII decoding

Now there's a potential problem: information leakage (of important info further up the stack). Programmers may not pay attention to sanitizing input like language config.

- %n: Assume then next argument is a pointer and then it writes the number of characters printed so far into that pointer.
- This can be abused by %n write to the return address and then overwrite it with the address of the shellcode.

How an exploit may look like for this is as follows:



1. Consume the 123 argument (%x)
2. Have the return address sitting in the beginning of the memory
3. Overwrite the RA value with the start of shellcode

There are some problems with this because on modern machines addresses are very large and it can be impractical to create a gigabyte-sized buffer. Instead we can just divide the problem up and write multiple 8 bit numbers

Example: "%243d" writes an integer with a field width of 243; "%n" will be incremented by 243.

Figure 7. The printf count increments by 243 with %243d. Shorthand

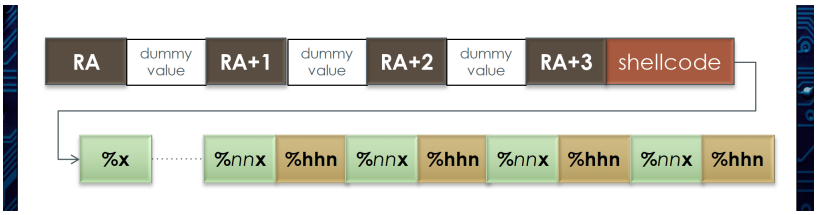


Figure 8. The gaps are there because

Dividing the problem into pieces; using %hhn and %nnx to write 8 bits at a time.

If the bytes being written must be written in decreasing order we can do this by structuring our pointers in a way that we write it in reverse order (don't need to start with LSB). Another option is

Note that this writes the printf counter into the pointer at the argument. This drastically decreases the buffer size needed

Subsection 2.4

Double-Free vulnerability

Freeing a memory location that is under the control of an attacker is an exploitable vulnerability

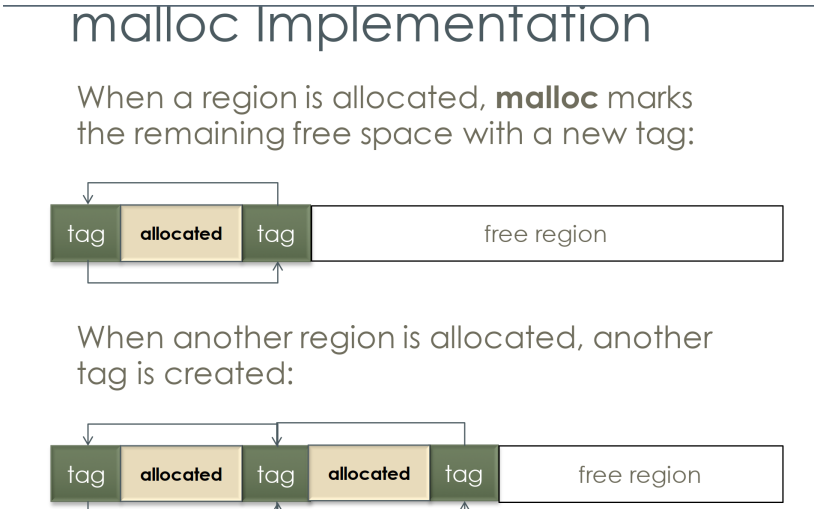
```
1  p = malloc(128);
2  q = malloc(128);
3  free(p);
4  free (q);
5  p = malloc(256);
6  // this is where the attack happens; the fake tag, shell code, etc
7  strcpy(p, attacker_string);
8  free(q);
```

Note that the `c` free function takes a reference (not necessarily a pointer) to the memory location to be freed. It does not change the value of the free'd pointer either.

malloc implementation

malloc maintains a doubly-linked list of free and allocated memory regions:

- Information about a region is maintained in a **chunk tag** that is stored just before the region
- Each chunk maintains:
  - A "free bit", indicating whether the chunk is allocated or free
  - Links to the next and previous chunk tags
- Initially when all memory is unallocated, it is in one free memory region



Malloc places a doubly-linked-list of chunk bits in memory to describe the memory allocated. `free` sets the free bit, does the various doubly linked list operations (looking at the pointer values of its neighbours in order to remove itself) and also tries to consolidate adjacent free regions. It assumes that it is passed the beginning of the allocated memory as well as go find the tag associated with the region (which is in a consistent place every time).

The attacker can drop fake tags into memory just ahead of the value we want to overwrite and then we can use the free() call to overwrite memory with the attacker's data. For example we can create a fake tag node with a prev and next pointer. We make the next pointer be the address of the return address. And then "prev" can contain the address of the start of our shellcode. So freeing on this fake tag will overwrite the return address with the shellcode start.

Comment

> Note that this is not possible with a single free if you are not able to write to negative memory indicies. The part that makes this attack works is that the double free allows the attacker to write a fake tag just before the next tag in a totally valid way. Doing this with a single free would also involve writing to memory that the program doesn't own. (recall: how the memory manager works)

Subsection 2.5

Other common vulnerabilities

The attacks we have seen have involved overwriting the return address to point to injecting code. Are there ways to exploit software without injecting code? Yes – return into `libc` i.e. use `libc`'s `system` library call which looks already like shell code. This can be accomplished with any of the exploits we have already talked about.

I.e.

- Change the return addresses to point to start of the system function
- Inject a stack frame on the stack
- Before return sp points to $system$
- System looks in stack for arguments
- System executes the command, i.e maybe a shell
- Function pointers

- Dynamic linking
- Integer overflows
- Bad bounds checking

2.5.1 Attacks without overwriting the return address

Finding return addresses is hard. So we can use other methods to inject code into the program.

- Function pointers: an adversary can just try to overwrite a function pointer
- An area where this is very common is with *dynamic linking*, i.e. functions such as *printf*.
- Typically both the caller of the 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 code in the library
- The dynamic linker performs this mapping with the procedure linkage table and the global offset table
  - GOT is a table of pointers to functions; contains absolute mem location of each of the dyn-loaded library functions
  - PLT is a table of code entries: onee per each library function called by program, i.e. sprintf@plt
  - Similar to a switch statement
  - Each code entry invoes the function pointer in the GOT
  - i.e. sprintf@plt may invoke jmp GOT[k] where k is the index of sprintf in the GOT
  - So if we change the pointers in the offset table we can make the program call our own code, i.e. with objdump.[3]

[3] *PLT/GOT always appears at a known location.*

2.5.2 Return-Oriented Programming

- An exploit that uses carefully-selected sequences of existing instructions located at the end of existing functions (gadgets) and then executes functions in an order such that these gadgets compose together to deliver an exploit.
- This can be done faster by seeding the stack with a sequence of return addresses corresponding to the gadgets and in the order we want to run them in.

Section 3

ECE353 Operating Systems

Subsection 3.1

Kernel Mode

3.1.1 ISAs and Permissions

There are a number of ISAs in use today; x86 (amd64), aarch64 (arm64), and risc-v are common ones. For purposes of this course we will study largely arm systems but will touch on the other two as well.

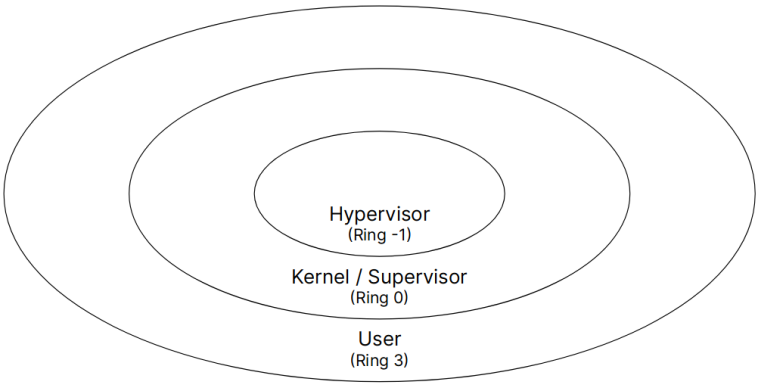


Figure 9. x86 Instruction access rings. Each ring can access instructions in its outer rings.

The kernel runs in, well, Kernel mode. **System calls** offer an interface between user and kernel mode[4].

The system call ABI for x86 is as follows:

Enter the kernel with a `svc` instruction, using registers for arguments:

- `x8` — System call number
- `x0` — 1st argument
- `x1` — 2nd argument
- `x2` — 3rd argument
- `x3` — 4th argument
- `x4` — 5th argument
- `x5` — 6th argument

This ABI has some limitations; i.e. all arguments must be a register in size and so forth, which we generally circumvent by using pointers.

For example, the `write` syscall can look like:

```
1 ssize_t write(int fd, const void* buf, size_t count);
2 // writes bytes to a file descriptior
```

3.1.2 ELF (Executable and Linkable Format)

- Aways starts with 4 bytes: `0x7F, 'E', 'L', 'F'`
- Followed byte for 32 or 64 bit architecture
- Followed by 1 byte for endianness

`readelf` can be used to read `ELF` file headers.
For example, `readelf -a $(which cat)` produces (output truncated)

[4] *Linux has 451 total syscalls*
Note: API (application programming interface), ABI (Application Binary Interface). API abstracts communication interface (i.e. two ints), ABI is how to layout data, i.e. calling convention

Most file formats have different starting signatures or magic numbers

```
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              DYN (Position-Independent
   ↪  Executable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x32e0
  Start of program headers:          64 (bytes into file)
  Start of section headers:          33152 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         13
  Size of section headers:           64 (bytes)
  Number of section headers:         26
  Section header string table index: 25
```

This output is followed by information about the program and section headers

`strace` can be used to trace systemcalls. For example let's look at the 168-byte hello-world example

```
0x7F 0x45 0x4C 0x46 0x02 0x01 0x01 0x00 0x00 0x00 0x00 0x00 0x00 0x00
 ↪  0x00 0x00
0x02 0x00 0xB7 0x00 0x01 0x00 0x00 0x00 0x78 0x00 0x01 0x00 0x00 0x00
 ↪  0x00 0x00
0x40 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
 ↪  0x00 0x00
0x00 0x00 0x00 0x00 0x40 0x00 0x38 0x00 0x01 0x00 0x40 0x00 0x00 0x00
 ↪  0x00 0x00
0x01 0x00 0x00 0x00 0x05 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
 ↪  0x00 0x00
0x00 0x00 0x01 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x01 0x00 0x00 0x00
 ↪  0x00 0x00
0xA8 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0xA8 0x00 0x00 0x00 0x00 0x00
 ↪  0x00 0x00
0x00 0x10 0x00 0x00 0x00 0x00 0x00 0x00 0x08 0x08 0x80 0xD2 0x20 0x00
 ↪  0x80 0xD2
0x81 0x13 0x80 0xD2 0x21 0x00 0xA0 0xF2 0x82 0x01 0x80 0xD2 0x01 0x00
 ↪  0x00 0xD4
0xC8 0x0B 0x80 0xD2 0x00 0x00 0x80 0xD2 0x01 0x00 0x00 0xD4 0x48 0x65
 ↪  0x6C 0x6C
0x6F 0x20 0x77 0x6F 0x72 0x6C 0x64 0x0A
```

Listing 2: Note: This is for arm cpus

If we run this then we see that the program makes a `write` syscall as well as a `exit_group`

```
execve (" ./ hello_world " , [ " ./ hello_world " ] , 0 x7ffd0489de40
  ↪  /* 46 vars */ ) = 0
write (1 , " Hello world \ n " , 12) = 12
exit_group (0) = ?
+++ exited with 0 +++
```

Note that these strings are not null-terminated (null-termination is just a `c` thing) because we don't want to be unable to write strings with the null character to it.

```
execve("./hello_world_c", ["./hello_world_c"], 0x7ffcb3444f60 /* 46 vars */) = 0
brk(NULL)                               = 0x5636ab9ea000
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=149337, ...}) = 0
mmap(NULL, 149337, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f4d43846000
close(3)                                = 0
openat(AT_FDCWD, "/usr/lib/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0000C"..., 832) = 832
lseek(3, 792, SEEK_SET)                 = 792
read(3, "\4\0\0\0\24\0\0\0\3\0\0\0GNU\0\201\336\t\36\251c\324"..., 68) = 68
fstat(3, {st_mode=S_IFREG|0755, st_size=2136840, ...}) = 0
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0)
  = 0x7f4d43844000
lseek(3, 792, SEEK_SET)                 = 792
read(3, "\4\0\0\0\24\0\0\0\3\0\0\0GNU\0\201\336\t\36\251c\324"..., 68) = 68
lseek(3, 864, SEEK_SET)                 = 864
read(3, "\4\0\0\0\20\0\0\0\5\0\0\0GNU\0\2\0\0\300\4\0\0\0\3\0\0", 32) = 32
```

Figure 10. A c hello world would load the stdlib before printing...

3.1.3 Kernel

The kernel can be thought of as a long-running program with a ton of library code which executes on-demand. Monolithic kernels run all OS services in kernel mode, but micro kernels run the minimum amount of servers in kernel mode. Syscalls are slow so it can be useful to put things in the kernel space to make it faster. But there are security reasons against putting everything in kernel mode.

3.1.4 Processes Syscalls

A process is like a combination of all the virtual resources; a "virtual GPU" (if applicable), memory (addr space), I/O, etc. The unique part of a struct is the PCB (Process Control Block) which contains all of the execution information. In Linux this is the `task`$_s truct which contains information about the process state, CPU registers, scheduling information, and so forth.$

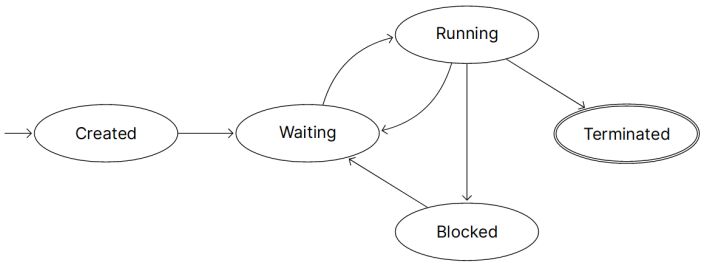


Figure 11. A possible process state diagram

These state changes are managed by the Process and OS[5] so that the OS scheduler can do its job. An example of where some of these states can be useful would be to free up CPU time while a process is in the Blocked state while waiting for IO. Process can either manage themselves (cooperative multitasking) or have the OS manage it (true multitasking). Most systems use a combination of the two, but it's important to note that cooperative multitasking is not true multitasking.

Context switching (saving state when switching between processes) is expensive. Generally we try to minimize the amount of state that has to be saved (the bare minimum is the registers). The scheduler decides when to switch. Linux currently uses the CFS[6].

In c most system calls are wrapped to give additional features and to put them more concretely in the userspace.

[5] *I think*

Process state can be read in /proc for linux systems.

[6] *completely fair scheduler*

```c
#include <stdio.h>
#include <stdlib.h>

void fini(void) {
  puts("Do fini");
}

int main(int argc, char **argv) {
  atexit(fini);
  puts("Do main");
  return 0;
}
```

Figure 12. Demonstration of c feature to register functions to call on program exit

```c
int main ( int argc , char * argv []) {
  printf ("I 'm going to become another process \n" );
  char * exec_argv [] = {" ls " , NULL };
  char * exec_envp [] = { NULL };
  int exec_return = execve ("/ usr / bin / ls " , exec_argv ,
  ↪  exec_envp );
  if ( exec_return == -1) {
    exec_return = errno ;
    perror (" execve failed " );
    return exec_return ;
  }
  printf (" If execve worked , this will never print \n" );
  return 0;
}
```

Listing 3: Demo of execve turning current program to ls (executes program, wrapper around exec syscall)

```c
#include <sys/syscall.h>
#include <unistd.h>

int main (){
  syscall(SYS_exit_group, 0);
}

```

Listing 4: An example of using a raw syscall system exit instead of c's exit()