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

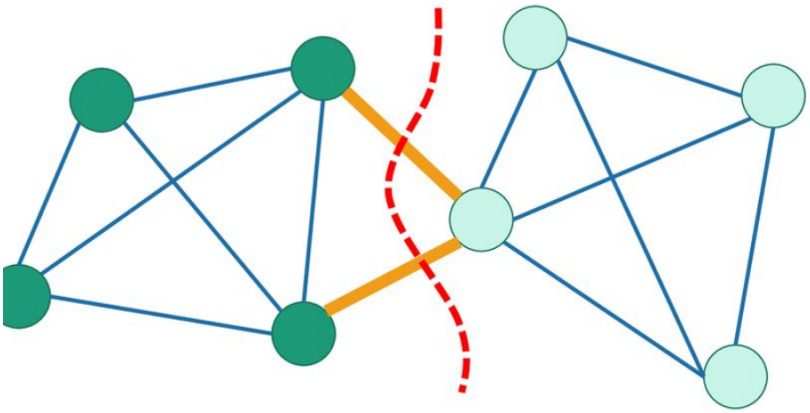


Figure 1. Example of global min-cut. Note that the global min-cut is not necessarily unique

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.

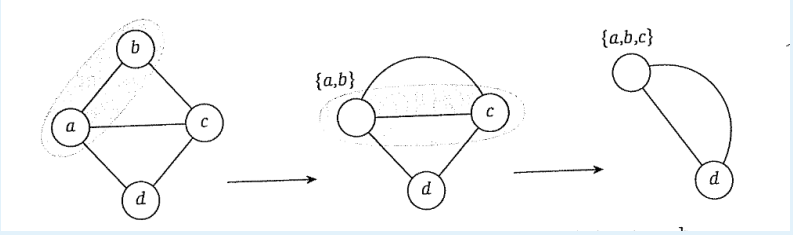


Figure 2. Example of a series of contractions

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 $(S_1, \hat{S_1}) =$ MIN-CUT(G')
6 $(S_2, \hat{S_2}) =$ MIN-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*

PROOF The intuition for this can be developed by drawing out a recursion tree for this problem. At each level the number of recursive call doubles, but the time it takes for each sub-call halves as well. This means that the total runtime for each level is n^2. As for the total time will just be $O(n^2 \log(n))$, since we know the height of the recursion tree to be $\log n$. More formally, the recursion may be described with

$$T(n) \leq 2T(\frac{n}{\sqrt{2}} + O(n^2)) \tag{1.9}$$

Which can [3] be solved with the master theorem.

[3] *I think?*

□

We may also want to understand the probability of success.

- We may deem a node in the recursion tree to be *successful* if it survives the contractions.
  - There must be a leaf node in a recursion tree that successfully produces a min-cut that corresponds to a min-cut, therefore there must also be a sequence of *successful* nodes from the root to said min cut.
- $P(d)$ as the probability that a node at depth d is successful, conditioned on it's ancestors being successful

Now, how can we find $P(h)$? I.e. the probability of the algorithm being successful on termination.

- Base case: $P(0) \geq \frac{1}{2}$ (will assume $= \frac{1}{2}$, worst case)
- Inductive step: $P(D) = \frac{1}{2}(1 - (1 - (P(d-1)))^2) = \frac{1}{2}(2P(d-1) - P(d-1)^2)$
  - At each level the probability of success is at least $\frac{1}{2}$, conditioned on the ancestors being successful.

What remains now is solving this recursion.

$$P(d) = P(d-1) - \frac{1}{2}P(d-1)^2 \tag{1.10}$$

The way of solving a non-linear recursion is to make a guess. We expect this to be on the order of

$$P(d) \geq \frac{1}{d} \tag{1.11}$$

And then as it turns out it's

$$P(d) = \frac{1}{d+2} \tag{1.12}$$

And then this can be checked by doing an induction proof

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

Subsection 1.3

Closest Pair Problem

- **Input**: A set P of n points in the plane
- **Output**

Approaches:

- Brute force $O * n^2$
- Divide and conquer (CLRS 33.4) $O(n \log n)$
- Rabin's Algorithm $O(n)$ expected time

RABINS-ALGORITHM(P)

```
1  Randomly order P as p_1, p_2 ... p_n
2  cp = {p_1, p_2}
3  Δ = dist(p_1, p_2)
4  for i = 3 to n
5      if ∃ q ∈ P_{i-1} = p_1 ... p_{i-1} s.t. dist(p, q) < Δ
6          cp = {p, q}
7          Δ = dist(p, q)
```

Considerations to make:

- How can we randomly order P from $n!$ possible orderings in a good time bound?. Note that it is possible to do it in $O(n)$ time (CLRS reference for later)
- What data structure to use for line 5? I.e. finding q such that $dist(p, q) < \Delta$
  - Note: take q that is closest to p_i in line 5 (algorithm is unclear as to pick p_1 or p_2)
  - This data structure must store a set $Q = P_{i-1} = p_1, p_2, p_{i-1}$ points and offer the following operations
    - Note: $\Delta(Q) =$
    - INSERT-FAST(p) inserts p into Q assuming $min\{dist(p, q) : q \in Q\} \geq \Delta Q$

* INSERT-SLOW(p) inserts p into Q even if $min\{dist(p,q) : q \in Q\} < \Delta Q$
* CHECK-CLOSEST(p) checks if $min\{dist(p,q) : q \in Q\} < \Delta$ and if so returns the closest point $q \in Q$ to P, otherwise returns NIL. Runs in $O(1)$ expected time

In more detail,

RABINS-ALGORITHM(P)

1 Randomly order P as $p_1, p_2 \dots p_n$
2 $cp = \{p_1, p_2\}$
3 $\Delta = dist(p_1, p_2)$
4 Initialize the data structure with $\{\}$
5 **else** $cp = \{p_1, q\}$ $\Delta = dist(p_1, q)$ INSERT-SLOW(p_1)

- This algo has runtime of $O(n^2)$ worst case. Line 1 is $O(n)$, and all the inserts run in $O(1)$ expected. In the event that we always INSERT-SLOW then it's $O(n^2)$ – but this doesn't happen very often, so realistically it's $O(n)$

How can we implement this data structure that enables this black magick? **Idea:** draw grid with side length $\frac{\Delta Q}{2}$. No two points in Q land in the same square. Hash grid squares.

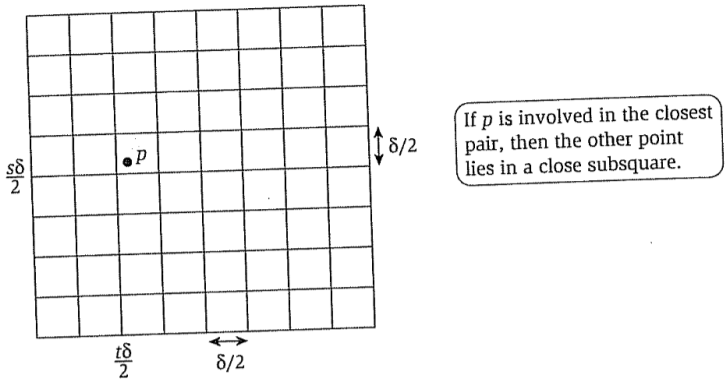


Figure 13.2 Dividing the square into size $\delta/2$ subsquares. The point p lies in the subsquare S_{st}.

- The longest distance in the grid is a diagonal along a square, which is $\frac{\Delta^2}{2}$
- In order to store Q we can just build a hashtable keyed by the grid square
  - For example for floating point Cartesian coordinates we may use use the floor function as the key to the hashtable.
- Insert-fast: just insert into the hashtable
- Insert-slow: must rebuild the data structure with new Δ
- Check-closest: Must look in the 25 squares around p for points closer than Δ. If there are more just return the closest.

1.3.1 Analysis

- Recall $P_i = \{p_1 \dots p_i\}$. Define (for $i \geq 3$) $Z_i = \begin{cases} 1 & \Delta(P_i) < \Delta(P_{i-1}) \\ 0 & \text{otherwise} \end{cases}$

  - This is a random event since the order of points is random. The first case represents a slow insert and the other a fast insert.
  - $T \leq n + \sum_{i=3}^{n}(1 + iZ_i)$: The runtime of the algorithm: n for the random init and then 1 for fast insert and iZ_i for a slow insert.
  - $E[T] \leq n + (n-2) + \sum_{i+3}^{n} i \cdot EZ_i$ (by linearity of expectation)
  - $= 2n - 2 + \sum_{i=3}^{n} i \cdot P(Z_i = 1)$

- Now, what's $P(Z_i) = 1$?
  - Let $\{p_i, p_k\}$ be a closest pair in P_i
  - If $Z_i = 1$ then p_i or p_k is p_i
  - $P(Z_i = 1) \leq \frac{2}{i}$ (There are two bad options out of i options)

Section 2

ECE568 Computer Security

Subsection 2.1

Refresher & Introduction

Comment I've found that the way that this course is organized does not lend itself well to well-organized headers and notes. Apologies for the train-of-thought style.

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

Note that the stack grows downwards and the heap grows upwards. Stack overflows can occur and can be a source of vulnerability.

GDB offers some tools for examining stacks

- `break`: create a new breakpoint

- `run`: start a new process
- `where`: list of current stack frames
- `up/down`: move between frames
- `info frame` display info on current frame
- `info args`: list function arguments
- `info locals`: list local variables
- `print`: display a variable
- `x` display contents of memory
- `fork`: Creates a new child process by duplicating the parent. The child has its own new unique process ID
- `exec`: Replaces the current process with a new process

The fork-exec technique is just a pair of `fork` and `exec` system calls to spawn a new program in a new process

2.1.1 Security Fundamentals

The three key components of security are:

- Confidentiality: the protection of data/resources from exposure, whether it be the content or the knowledge that the resource exists in the first place. Usually via organizational controls (security training), access rules, and cryptography.
- Integrity: Trustworthiness of data (contents, origin). Via monitoring, auditing, and cryptography.
- Availability: Ability to access/use a resource as desired. Can be hard to ensure; uptime, etc...

Together they form an acryonym: CIA. A system is considered secure if it has all three of these properties for a given time. The strength of cryptographic systems can be evaluated by the number of bits of entropy or their complexity. For example, a 128-bit key has 2$\hat{}$128 possible values. This would take a lot of time to break, and a 256-bit key even longer. Availability is harder to measure quantitatively and is instead traditionally measured qualitatively. For example, a system may be available 99.9% of the time. But this doesn't really measure w.r.t security.

Some security terms:

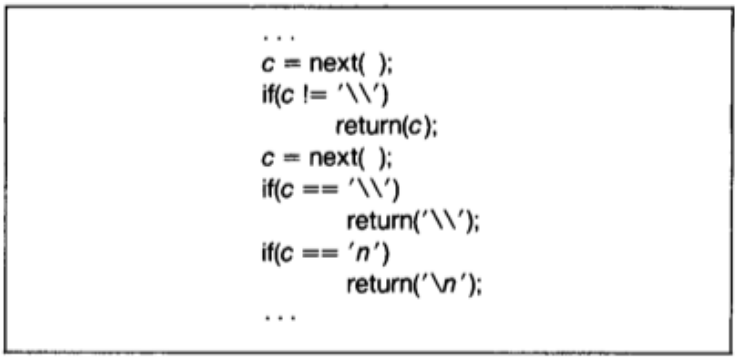
- Another security concept is the **threat**, or any method that can breach security.
- An exercise of a threat is called an **exploit** and a successful exploit causes the system to be compromised. Common threats include internet connections/open ports.
- **Vulnerabilities** are flaws that that weaken the security of a system and can be difficult to detect. For example an unchecked string copy can cause a buffer overflow and allow an attacker to execute arbitrary code
- **Compromises** are the intersection between threats and Vulnerabilities, i.e. when an attacker matches a threat with a vulnerability (i.e. matching a tool in the attacker's arsenal with a weakness)
- **Trust** : How much exposure a system has to an interface. For example a PC might have a lot of trust in the user.

The leading cause of computer security breaches are humans. We are prone to making mistakes. A general trade-off exists when designing secure systems for humans; the more secure a system becomes the less usable it tends to be. One way of measuring the quality of a security system is how secure it is while maintaining usability

2.1.2 Reflections on Trusting Trust

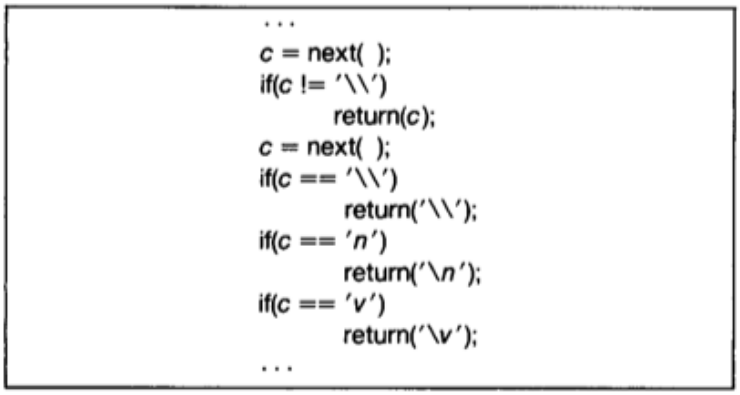
Comment

> **Reflections on Trusting Trust** is a paper by Ken Thompson that discusses the trust and security in computing. Cool short read.



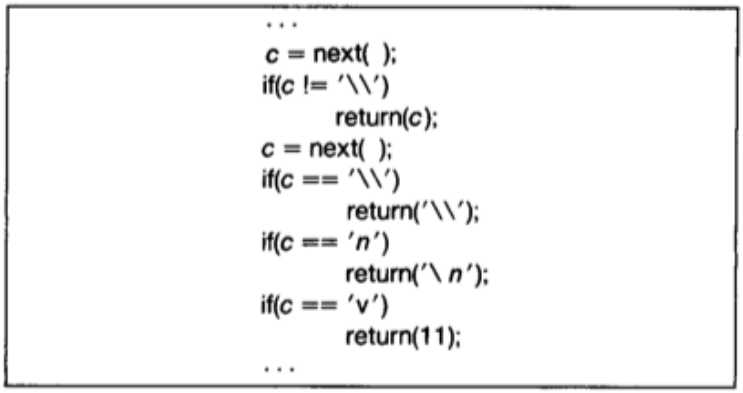
```
...
c = next( );
if(c != '\\')
        return(c);
c = next( );
if(c == '\\')
        return('\\');
if(c == 'n')
        return('\n');
...
```

FIGURE 2.2.



```
...
c = next( );
if(c != '\\')
        return(c);
c = next( );
if(c == '\\')
        return('\\');
if(c == 'n')
        return('\n');
if(c == 'v')
        return('\v');
...
```

FIGURE 2.1.



```
...
c = next( );
if(c != '\\')
        return(c);
c = next( );
if(c == '\\')
        return('\\');
if(c == 'n')
        return('\ n');
if(c == 'v')
        return(11);
...
```

FIGURE 2.3.

Figure 3. Teaching a compiler what the "\v" sequence is. We may add a statement to return the ascii encoding of \v (11), compile the compiler, and then use it to compile a program that knows what \v is.

. We may then alter the source to be like Figure 2.3 without any mention of \v but still compile programs with \v just fine.

```
compile(s)
char *s;
{
        ...
}
```

FIGURE 3.1.

```
compile(s)
char *s;
{
        if(match(s, "pattern")) {
                compile("bug");
                return;
        }
        ...
}
```

FIGURE 3.2.

```
compile(s)
char *s;
{
        if(match(s, "pattern1")) {
                compile ("bug1");
                return;
        }
        if(match(s, "pattern 2")) {
                compile ("bug 2");
                return;
        }
        ...
}
```

FIGURE 3.3.

Next, consider the above scenario where we insert a login Trojan to insert backdoors into code matching the unix login function. We may then compile the c compiler to do just that, and then change the source to what it should look like without the Trojan. Compiling the compiler one more time will now produce a compiler binary that looks completely innocent but will reinsert the Trojan wherever it can.

The moral of the story is that you can't trust code that you didn't totally create yourself. But it's awfully difficult to use only code written by oneself. So take security seriously.

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.

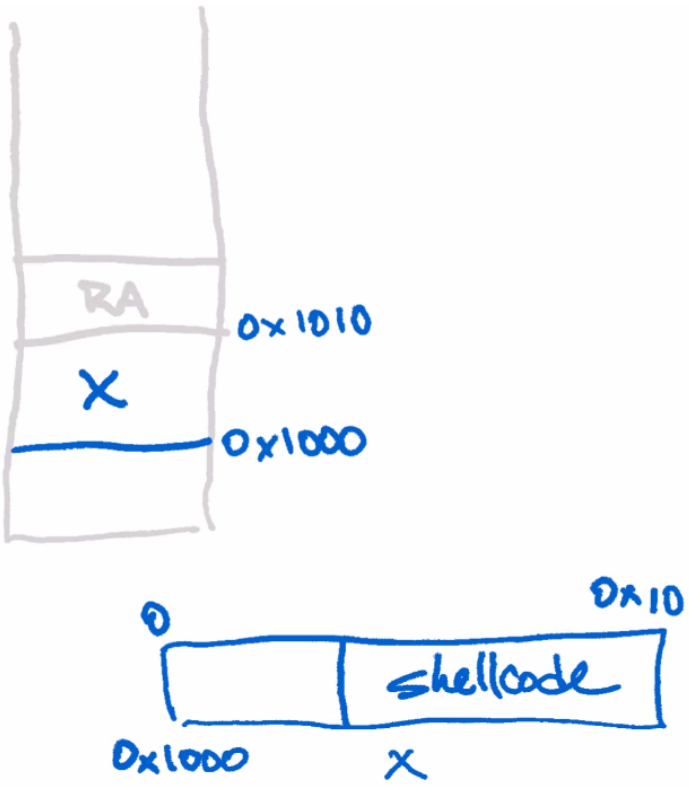


Figure 4. Bufferoverflow to write to the return address. Shellcode is a sequence of instructions that is used as the payload of an attack. It si called a shellcode because they commonly are used to start a shell from which the attacker can do more.

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.

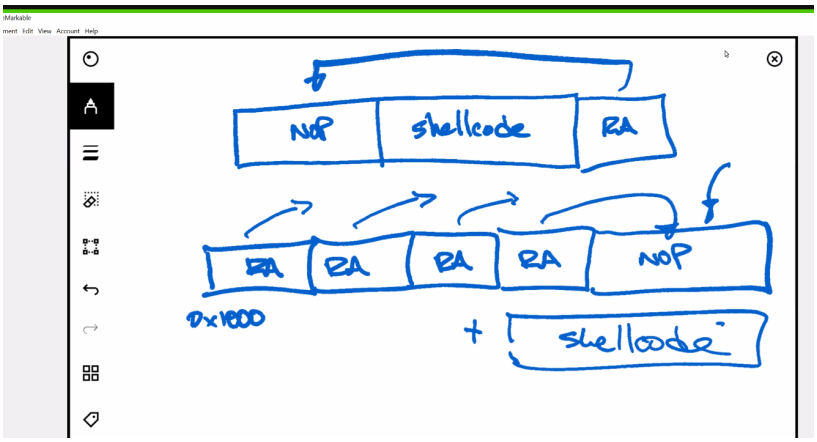


Figure 5. Can create a RA sled with a NOP leading to shellcode and then try it from e.g. 0x1000, 0x2000 and so forth to find where to attack from.

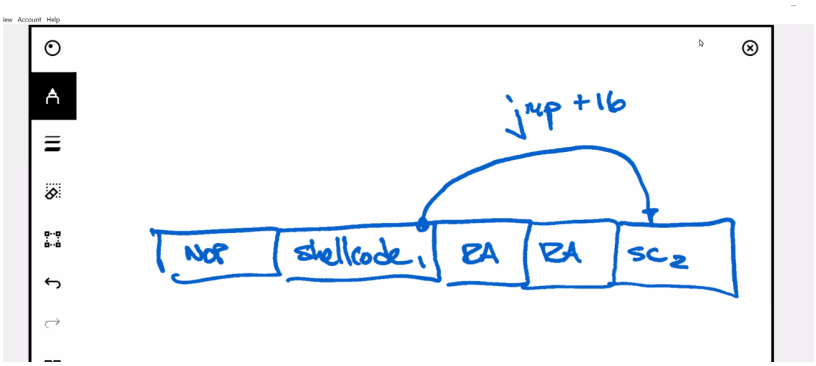


Figure 6. Another technique may involve placing shellcode all over the place, of which each one may be a valid entrypoint into the shellcode.

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
}
```

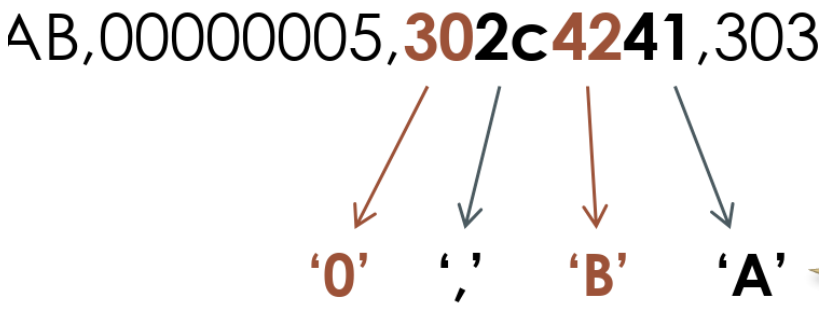
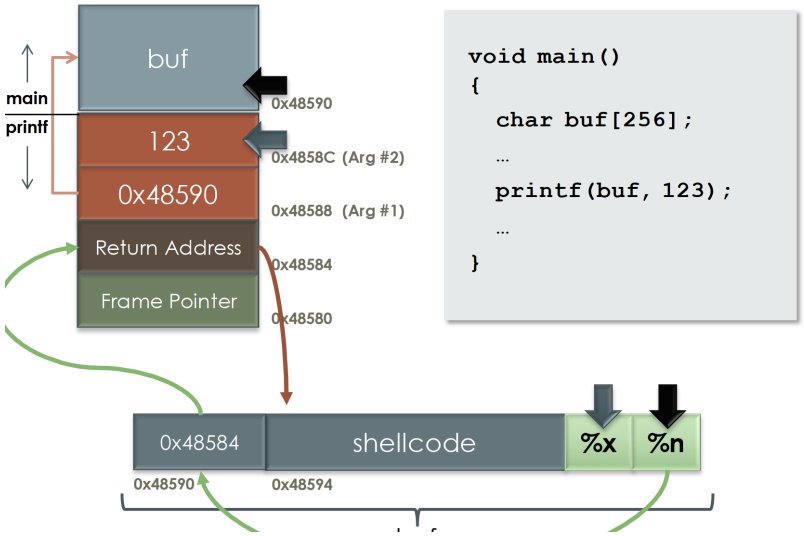


Figure 7. ASCII decoding

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 8. The printf count increments by 243 with %243d. Shorthand

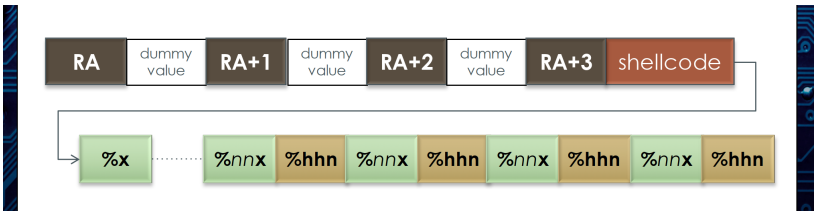


Figure 9. The gaps are there because

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

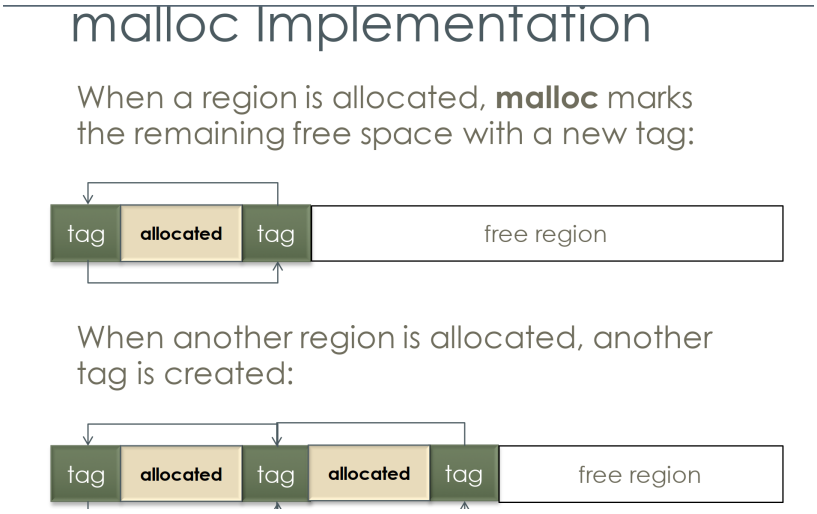
```
1  p = malloc(128);
2  q = malloc(128);
3  free(p);
4  free (q);
5  p = malloc(256);
6  // this is where the attack happens; the fake tag, shell code, etc
7  strcpy(p, attacker_string);
8  free(q);
```

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.[4]

[4] *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.

Subsection 2.6

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.

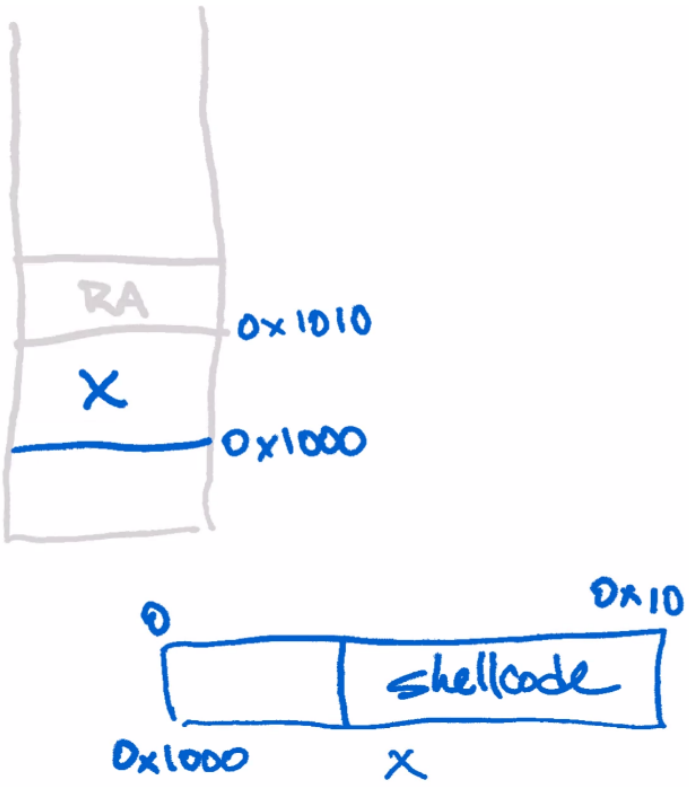


Figure 10. Bufferoverflow to write to the return address. Shellcode is a sequence of instructions that is used as the payload of an attack. It si called a shellcode because they commonly are used to start a shell from which the attacker can do more.

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.

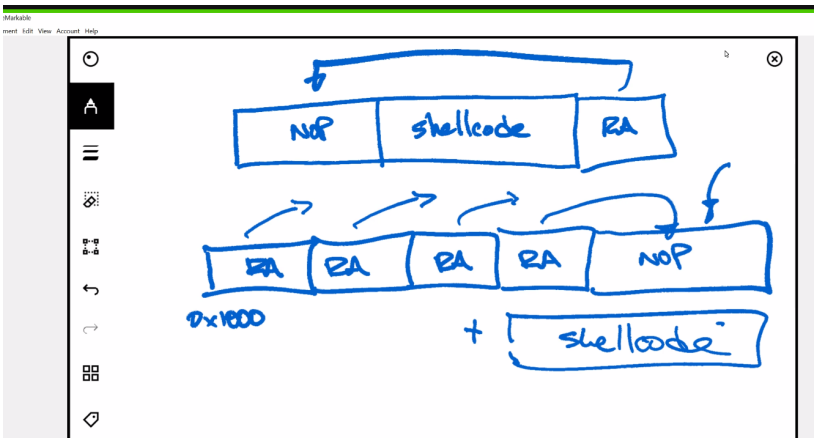


Figure 11. Can create a RA sled with a NOP leading to shellcode and then try it from e.g. 0x1000, 0x2000 and so forth to find where to attack from.

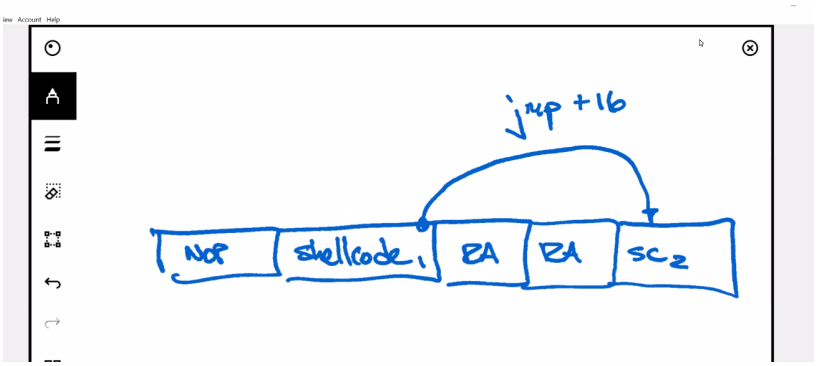


Figure 12. Another technique may involve placing shellcode all over the place, of which each one may be a valid entrypoint into the shellcode.

Subsection 2.7

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
}
```

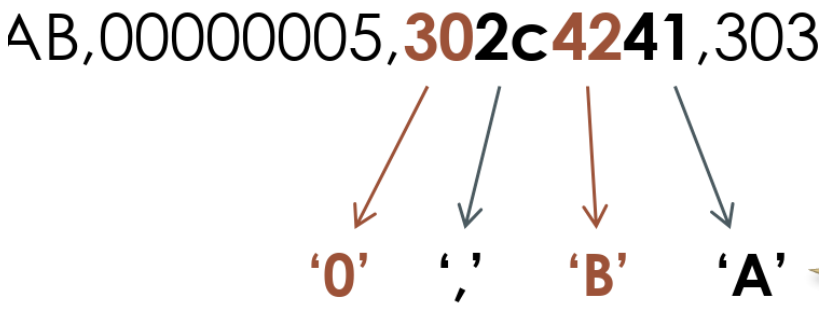
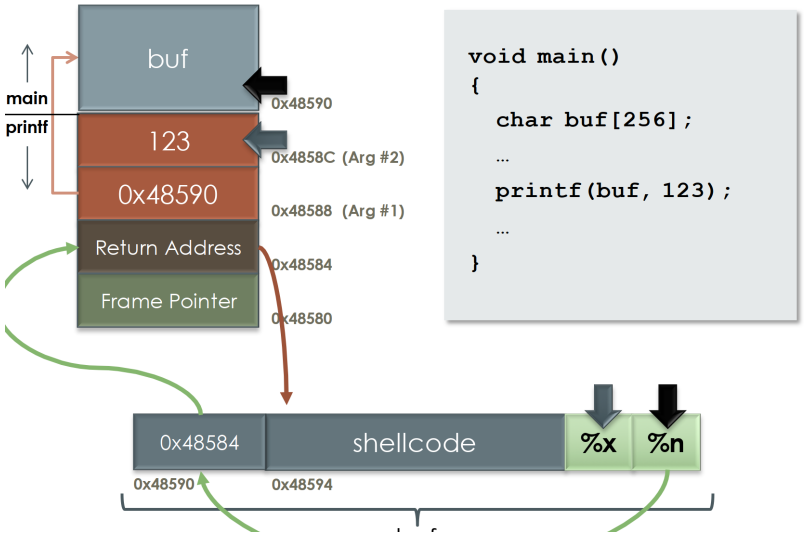


Figure 13. ASCII decoding

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 14. The printf count increments by 243 with %243d. Shorthand

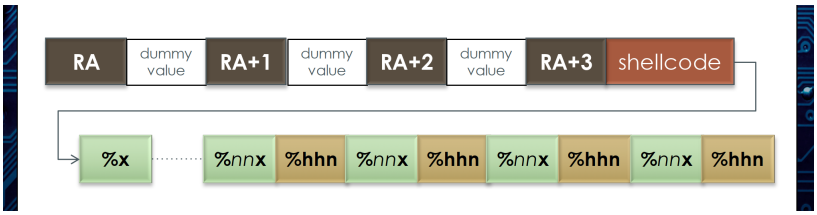


Figure 15. The gaps are there because

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 2.8

Double-Free vulnerability

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

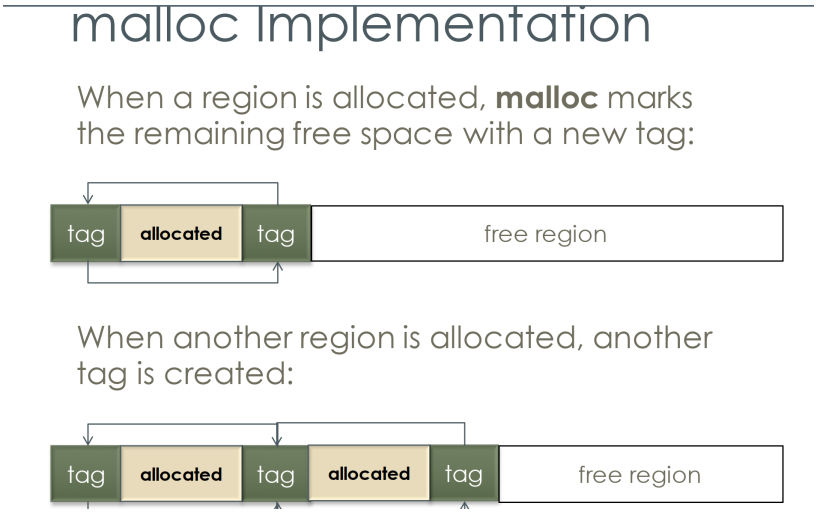
```
1  p = malloc(128);
2  q = malloc(128);
3  free(p);
4  free (q);
5  p = malloc(256);
6  // this is where the attack happens; the fake tag, shell code, etc
7  strcpy(p, attacker_string);
8  free(q);
```

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 2.9

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

2.9.1 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.[5]

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

2.9.2 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.

2.9.3 Deserialization attacks

- Serialization is the process of transforming objects into a format that can be stored or transmitted over a network, i.e. to/from JSON.
- The attacker knows that the library has a vulnerability in the deserialization process and they can exploit it by passing carefully created data to it.

2.9.4 Integer overflows

- A server processes packets of variable size
- First 2 bytes of the packet store the size of the packet to be processed
- Only packets of size 512 should be processed
- Problem: what if we end up overflowing the integer with a negative value which would cause memcpy to copy over a lot more memory than intended.

```
char* processNext(char* strm){
    char buf[512];
    short len = *(short*)strm; // note that by default these are
    ↪ signed
    if (len <= 512) {
        memcpy(buf, strm, len); // note that the 3rd arg of memcpy is
    ↪ an unsigned int
        process(buf);
        return strm + len;
    } else {
        return -1
    }
}
```

2.9.5 IoT

Subsection 2.10

Case Study: Sudo

A common program attackers target are programs that regular users can run in order to take on elevated privileges. In `unix` systems one such program is `sudo`, for which vulnerability CVE-2021-3156 was discovered in 2021 after lying in there for over 10 years.

- `sudo` will escape certain characters such as "
- Someone introduced debug logic called user_args and then copies in the contents of `argv`, while un-escaping meta-characters
- Bug: if any command-line arg ends in a single backslash, then the null-terminator gets un-escaped and then `user_args` keeps copying out of bounds characters onto the stack
- I.e. `sudoedit -s '\' $(perl -e print "A"x1000$)`
- Attacker controls the size of `user_args` buffer they overflow. Can control size and contents of the overflow itself; last command-line argument is followed by the environment variables
- Had many exploit options
  - Overwrite next chunk's memory tag (same as use-after-free)
  - Function pointer overwrite one of sudo's functions
  - Dynamically-linked library overwrite
  - Race condition a temp file sudo creates
  - Overwrite the string "usr/bin/sendmail" with the name of another executable, maybe a shell

Subsection 2.11

Case Study: Buffer overflow in a Tesla

`ConnMann` (Connection Manager) is a lightweight network manager used in many embedded systems, i.e. nest thermostats and Teslas for that manager.

In this particular vulnerability the attacker took advantage of the DNS protocol. DNS responses include a special encoding for the hostnames which help the receiver parse the response and allocated appropriately sized buffers. For example www.google.com is encoded as

3www6google3com. This response also often contains a lot of repetitive information, so there is some compression is used in the encoding as well. The one we're interested in here is the compression of names by encoding them as a special "field length" of 192 followed by the offset of the other copy of the name – which enables repetitions to be encoded as 2 bytes.

CVE-2021-26675 was reported by Tesla in 2021 as a bug in ConnMan which allows an malicious DNS reply to uncompress into a large string that can overflow an internal buffer. This means that a remote attacker who can control or fake a DNS response could perform a buffer overflow on ConnMan – which runs with root privileges.

```
static gboolean listener_event(...)
{
        GDHCPClient *dhcp_client = user_data;
        struct sockaddr_in dst_addr = { 0 };
        - struct dhcp_packet packet;
        + struct dhcp_packet packet = { 0 };
        struct dhcpv6_packet *packet6 = NULL;
        ...
```

Figure 16. ConnMan doesn't initialize the dhcp_packet struct to 0, which can cause it to leak stack values to a remote attacker (but here they must be on the same subnet as the victim). This vulnerability can be difficult to detect since nobody checks if things are zero in the tests.

Comment

So you want to be a hack a tesla?

- Look at the situation; see what kind of protocols being used, etc. Get excited if it uses something old and inane
- Look at the data coming in and our, especially if there's any extra going in or out
- Use fuzzing tools
- Get a sense of what they are expecting us to do as well as what are ways that we can break that example. For example is the only verification just some client-side JavaScript?
- Break stuff

Subsection 2.12

Fault Injection Attacks

We make a lot of assumptions about how the underlying systems work. For example proper CPU operation. Fault injection attacks take advantage of these assumptions by injecting faults into the system, often at the hardware level.

For example: proper pipelined CPU operation depends on stable power and clock inputs. If the glitch duration is longer than the time it takes to increment the PC and shorter than the instruction fetch time, then we can start to see a special case: instruction skipping or instruction corruption.p

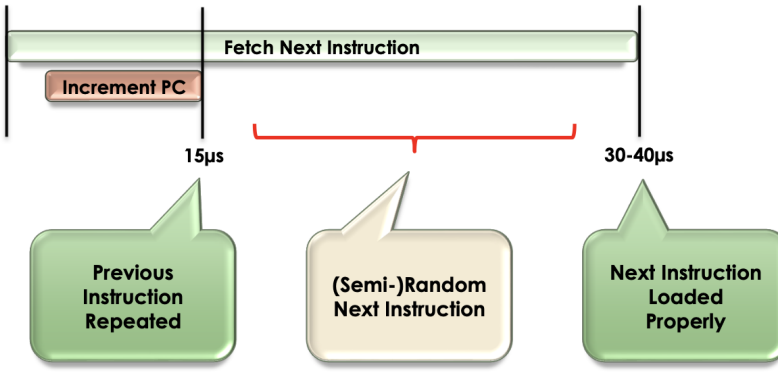


Figure 17. With some careful timing we can cause the CPU to skip or repeat an instruction.

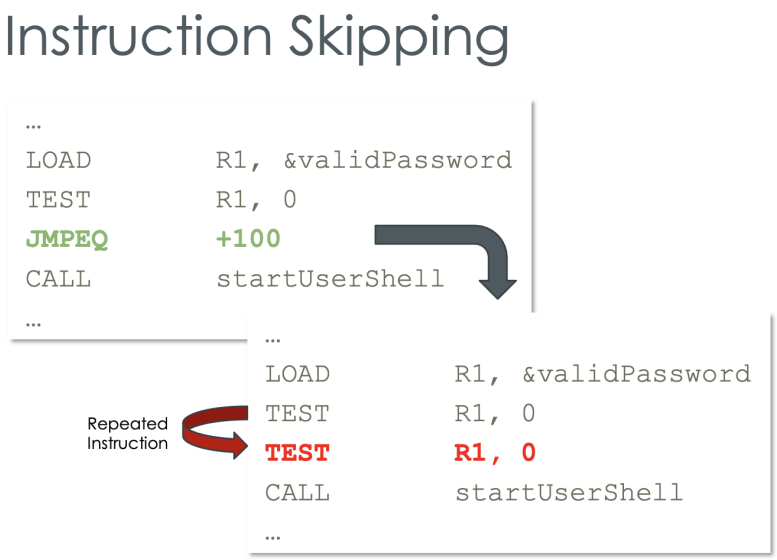
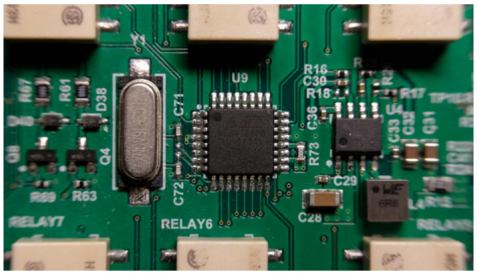


Figure 18. An example of where this can be useful: skipping the JMP instruction of an IF statement

2.12.1 Hardware Demo

Consider this simple program that checks a text buffer for a password and then logs you in if it's correct



ATMega328P Microcontroller

- Low-powered microprocessor + flash memory
- One application, no traditional operating system
- Common in **IoT** (home automation) and **embedded** (industrial control) applications

```
login: root
Password: password

Login incorrect.

login: root
Password: s3cr3tP4ssw0rd&:-)@#

Last login: Wed Dec 31 12:34:56 2025 from 127.0.0.1
root@iotvictim:~#
```

```
…
bool passwordIsValid =
       !strcmp("s3cr3tP4ssw0rd&:-)@#", buffer);

if ( !usernameIsValid || !passwordIsValid ) {

       // Login incorrect
       Serial.println("\n\nLogin incorrect.");
L: SKIP THIS!  return;
}

// Login correct
…
```

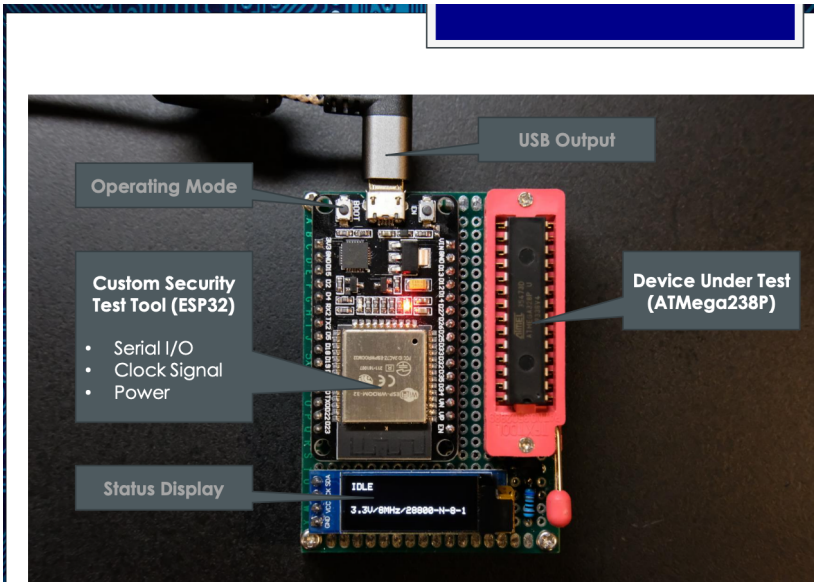


Figure 19. The ATMega238P hooked up to a custom security test tool built on top of a ESP32

Our attack is to use a **clock glitch**[6] at the time of the return instruction. Finding the time of the return instruction is a bit tricky but we can just sweep across a range of times.

[6] *A series of very brief and rapid clock pulses*

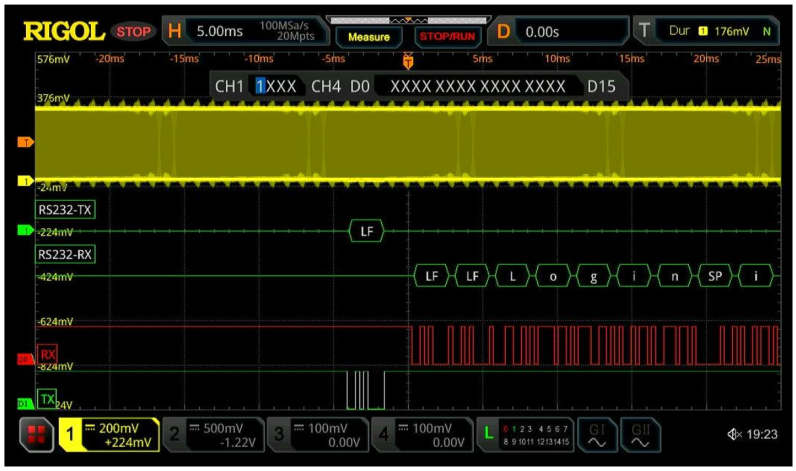


Figure 20. Looking at the oscilloscope to show the sweeped input

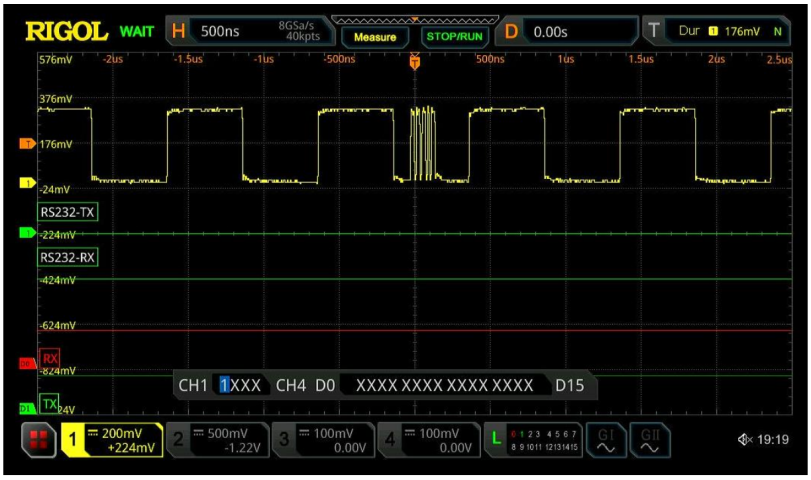


Figure 21. Note crazy clock pulses which we try to line up with the return instruction. If we have a really good chip we can try it with only 1 pulse, but here we use 5 pulses because we're on a cheaper chip. We also don't happen to care too much about whether or not if we disrupt too many of the other instructions.

Another attach that is a bit easier to use is the **power glitch**: instead of not giving it enough time for the fetch to happen we take away the nice voltage going to the chip right at the instruction execution time.

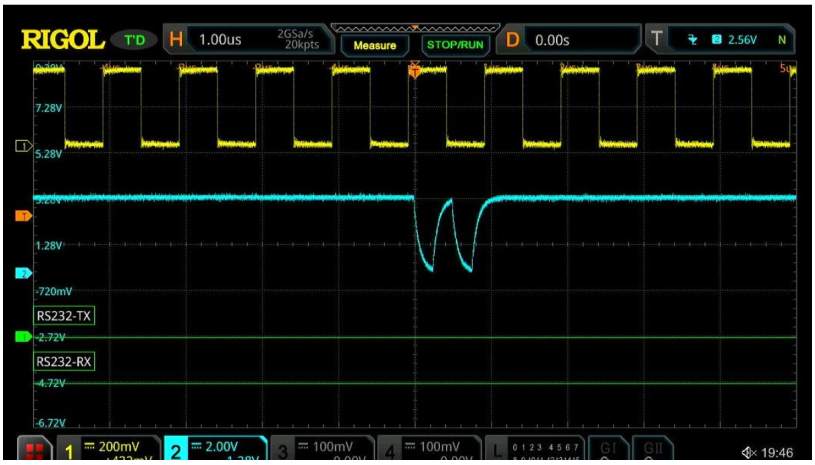


Figure 22. A power glitch attack disrupts the fetch instruction or the decode

Defense for these attacks is to disallow physical access to the chip. For the power one it can be as simple as adding a little capacitor to the power supply to smooth it out. Likewise, there are many ways to cause controlled circuit malfunctions: lasers, strobes, EM pulses, etc.

Subsection 2.13

Reverse Engineering

Reverse Engineering, or the act of analyzing a product in order to learn something about its design which its creator wanted to keep secret. It is a legally complicated, but generally it's ok for the purposes of achieving interoperability (but not for circumventing DRMs).

```
unsigned int printhelloworld() {
    printf("Hello World!");
    return 5;
}
int main (int argc, char *argv[]) {
    unsigned int result = 0;
    result = printhelloworld();
    if (result == 4) {
        printf("super secrete string\n");
    }
    return 0;
}
```

Figure 23. Passing the binary compiled from the above code to a disassembler

With the disassembled binary we know where the instruction for the if statement we were curious about lives, so we can then just use hexedit to change the bits at that JMP to a NOP to print out the super secret string.

Subsection 2.14

Buffer Overflow Defenses

- Audit code rigorously
- Use a type-safe language with bounds checking (Java, C#, rust)
- However, this is not always possible due to legacy code, performance, etc.
- Defending against stack smashing
  - Stackshield: put return addresses on a seperate stack with no other data buffers there
  - Stackguard: a random canary value is placed just before the RA on a function call. If the canary value changes, the program is halted. This can be enabled via a flag on most modern compilers.
- Third-party libc i.e. libsfae which doesn't allow for '%n' in format strings
- Address space layout randomization: maps the stack of each process at a randomly selected location with each invocation, so that an attacker will not be able to easily guess the target address. GCC does do this by default.

If we really sit down and think about it, it's basically impossible to defend against all attacks. It's easy to make a mistake and end up with a vulnerability. Certain vulnerabilities can be avoided by using safer languages, but the only real defense is to be aware and careful. One approach is what the aerospace industry does, i.e. the swiss cheese model [7]

[7] *if we stack a lot of hole-y cheese on top of each other it will be opaque*

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.

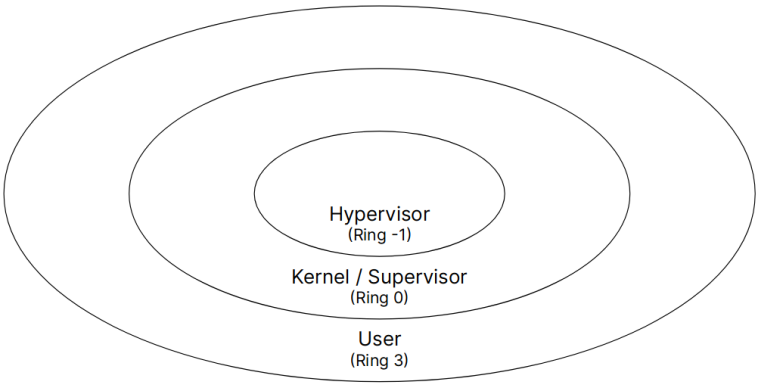


Figure 24. x86 Instruction access rings. Each ring can access instructions in its outer rings.

The kernel runs in, well, Kernel mode. **System calls** offer an interface between user and kernel mode[8].

[8] *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

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)

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
```

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

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

```
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 25. 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_struct` which contains information about the process state, CPU registers, scheduling information, and so forth.

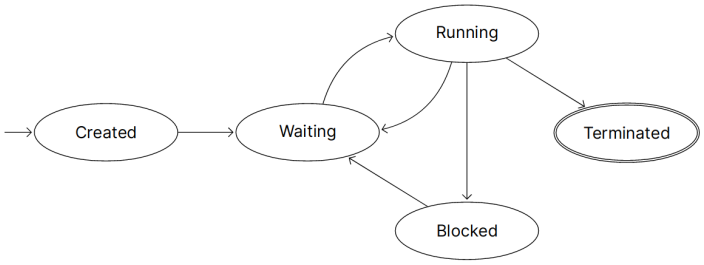


Figure 26. A possible process state diagram

These state changes are managed by the Process and OS[9] 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.

[9] *I think*

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[10].

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

[10] *completely fair scheduler*

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

```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 27. 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()

Subsection 3.2

Fork, Exec, And Processes

- `fork` creates a new process which is a copy of the current process. Everything is exactly the same except for the PID in the child and PID in the parent.
  - Returns -1 on error, 0 in the child process, and the pid of the child in the parent process
- `exec` replaces the current process with a new one
  - Returns -1 on error

Process states:

- The CPU is responsible for *scheduling* processes, so there can be >1 process per core.
- Maintaining the parent-child relationship
  - Parent is responsible for the child
  - This usually works; the parent can wait for the child to finish. But what if the parent crashes, etc?

- Zombie: a process that has finished but has not been cleaned up by its parent. This can be a problem because the process is still using resources. The OS has to keep a zomibe process until it's acknowledged. To avoid zombie build-up the OS can signal the parent process (over IPC) to acknowledge the child. (The parent can ignore it)
- Orphan: a process that has no parent. This can happen if the parent crashes. The OS can adopt the orphan and make it a child of the init process which can keep onto them or kill them as needed.

```c
int main(int argc, char *argv[]) {
    pid_t pid = fork();
  if (pid == -1) {
    int err = errno; perror("fork failed"); return err;
  }
  if (pid == 0) {
    printf("Child parent pid: %d\n", getppid());
    sleep(2);
    printf("Child parent pid (after sleep): %d\n", getppid());
  }
  else {
    sleep(1); }
  return 0; }
```

Listing 5: orphan example: parent exits before child and `init` has to clean up

Subsection 3.3

IPC

Reading and writing files is a form of IPC. For example, a simple process could write everything it reads, i.e this facsimile of the `cat` program

Standard file descriptors: 0 = stdin, 1 = stdout, 2 = stderr

```c
int main() {
  char buffer[4096];
  ssize_t bytes_read;
  // read (see man 2 read) reads from a file descriptor
  // can't assume always successful; see from `man errno`
  //   Nearly all of the system calls provide an error number in the external variable errno, which is defined as: extern int errno. Refer to man pages for what each errno means.

  while ((bytes_read = read(0, buffer, sizeof(buffer))) > 0) {
    ssize_t bytes_written = write(1, buffer, bytes_read);
    if (bytes_written == -1) {
      int err = errno;
      perror("write");
      return err;
    }
    assert(bytes_read == bytes_written);
  }
  if (bytes_read == -1) {
    int err = errno;
    perror("read");
    return err;
  }
  assert(bytes_read == 0);
  return 0;
}
```

Another way of IPC is using signals. Common signals include

- SIGINT (Ctrl-C)
- SIGKILL (kill -9)
- EOF (Ctrl-D)

A signal pauses (interrupts) your program and then runs the signal handler. Process can be interrupted at any point in execution, and the process will resume after the signal handler finishes.

```c
void handle_signal(int signum) {
  printf("Ignoring signal %d\n", signum);
}

void register_signal(int signum)
{
  struct sigaction new_action = {0};
  sigemptyset(&new_action.sa_mask);
  new_action.sa_handler = handle_signal;
  if (sigaction(signum, &new_action, NULL) == -1) {
    int err = errno;
    perror("sigaction");
    exit(err);
  }
}

int main(int argc, char *argv[])
{
  if (argc > 2) {
    return EINVAL;
  }

  if (argc == 2) {
    close(0);
    int fd = open(argv[1], O_RDONLY);
    if (fd == -1) {
      int err = errno;
      perror("open");
      return err;
    }
  }

  register_signal(SIGINT);
  register_signal(SIGTERM);

  char buffer[4096];
  ssize_t bytes_read;
  while ((bytes_read = read(0, buffer, sizeof(buffer))) > 0) {
    ssize_t bytes_written = write(1, buffer, bytes_read);
    if (bytes_written == -1) {
      int err = errno;
      perror("write");
      return err;
    }
    assert(bytes_read == bytes_written);
  }
  if (bytes_read == -1) {
    int err = errno;
    perror("read");
    return err;
  }
  assert(bytes_read == 0);
  return 0;
}
```

- `register_signal` sets a bunch of things such that we can handle the signal i.e. execute a function when a signal occurs. In this program we register SIGINT and SIGTERM with the kernel to execute `handle_signal`.
- This will still fail on ctrl-c because the read system call can error out

```c
ssize_t bytes_read;
  while ((bytes_read = read(0, buffer, sizeof(buffer))) != 0) {
    if (bytes_read == -1) {
      if (errno == EINTR) {
        continue;
      }
      else {
        break;
      }
    }
    ssize_t bytes_written = write(1, buffer, bytes_read);
    if (bytes_written == -1) {
      int err = errno;
      perror("write");
      return err;
    }
    assert(bytes_read == bytes_written);
  }
  if (bytes_read == -1) {
    int err = errno;
    perror("read");
    return err;
  }
  assert(bytes_read == 0);
  return 0;
}
```

- This snippet checks errno. and trys read again. Then the program is able to handle ctrl-c.
- This program can still get killed by kill -9 since it doesn't handle SIGKILL.
- Let's say we register $SIGKILL$ with the kernel to execute `handle_signal`. This will not work because you aren't allowed to ignore SIGKILL (-9).

Another thing we're interested in is to find out when a process is done. This can be polling on `waitpid`[11]

[11] *wait for process termination*

```c

int main() {
  pid_t pid = fork();
  if (pid == -1) {
    return errno;
  }
  if (pid == 0) {
    sleep(2);
  }
  else {
    pid_t wait_pid = 0;
    int wstatus;

    unsigned int count = 0;
    while (wait_pid == 0) {
      ++count;
      printf("Calling wait (attempt %u)\n", count);
      wait_pid = waitpid(pid, &wstatus, WNOHANG);
    }

    if (wait_pid == -1) {
      int err = errno;
      perror("wait_pid");
      exit(err);
    }
    if (WIFEXITED(wstatus)) {
      printf("Wait returned for an exited process! pid: %d, status:
↪  %d\n", wait_pid, WEXITSTATUS(wstatus));
    }
    else {
      return ECHILD;
    }
  }
  return 0;
}
```

Alternatively, we should use interrupts

Note: interrupt handlers run to completion. But an interrupt handler may occur while another interrupt handler is running, so execution must be passable and state managed accordingly

```c
void handle_signal(int signum) {
  if (signum != SIGCHLD) {
    printf("Ignoring signal %d\n", signum);
  }


  printf("Calling wait\n");
  int wstatus;
  pid_t wait_pid = wait_pid = waitpid(-1, &wstatus, WNOHANG);
  // Here in our interrupt (signal) handler) we check for SIGCHILD and then waitpid the child if applicable
  if (wait_pid == -1) {
    int err = errno;
    perror("wait_pid");
    exit(err);
  }
  if (WIFEXITED(wstatus)) {
    printf("Wait returned for an exited process! pid: %d, status: %d\n", wait_pid, WEXITSTATUS(wstatus));
  }
  else {
    exit(ECHILD);
  }
  exit(0);
}


void register_signal(int signum) {
  struct sigaction new_action = {0};
  sigemptyset(&new_action.sa_mask);
  new_action.sa_handler = handle_signal;
  if (sigaction(signum, &new_action, NULL) == -1) {
    int err = errno;
    perror("sigaction");
    exit(err);
  }
}

int main() {
  register_signal(SIGCHLD);

  pid_t pid = fork();
  if (pid == -1) {
    return errno;
  }
  if (pid == 0) {
    sleep(2);
  }
  else {
    while (true) {
      printf("Time to go to sleep\n");
      sleep(9999);
    }
  }
  return 0;
}
```

On a RISC-5 CPU there are three terms for interrupts:

- Interrupt: by external hardware and handled by kernel
- Exception: triggered by an instruction, kernel handles though process can optionally handle
- Trap: transfer of control of a trap handler by either an exception or interrupt. Syscall is a requested trap

Subsection 3.4

Pipe

Definition 3

```c
int pipe(int pipefd[2]);
```

Returns 0 on success, -1 on failure (and sets errno). Forms a one-way communication channel with 2 file descriptors; 0 for reading and 1 for writing. The pipe is unidirectional.

Subsection 3.5

Basic Scheduling

- A pre-emptive resource can be taken and used for something else; i.e. CPU. Shared via scheduling
- A non-pre-emptive resource cannot be taken and used for something else; i.e. I/O. Shared via alloc/dealloc or queuing. Note that some parallel or distributed systems may allow you to allocate a CPU
- Dispatcher: responsible for context switching. Scheduler: deciding which processes to run
- Non-preemptible processes must run until completion, so the scheduler can only make a decision on termination.
- Pre-emptive allows the OS to run scheduler at will.
- Schedulers seek to minimize waiting time and maximize cpu utilization/throughput – all while giving each process the same percent of CPU time.

Definition 4 FCFS (First come first served) is a scheduling algorithm that runs the process that arrives first. Processes are stored in a queue in arrival order. This has the downside of potentially introducing long wait times if longer tasks arrive before shorter ones.

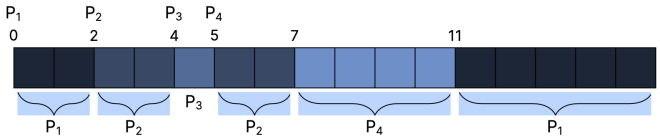
Definition 5 SJF (Shortest job first): schedule the job with the shorts execution time first. Though it's optimal at minimizing average wait times, since we don't know how long each process takes it may not be practically optimal. It also has the downside of potentially starving longer jobs.

Definition 6 SRTF (Shortest Remaining Time First): schedule the job (with pre-emptions now) with the shortest remaining time. This optimizes the average waiting time.

Consider the same processes and arrival times as SJF:

Process	Arrival Time	Burst Time
P_1	0	7
P_2	2	4
P_3	4	1
P_4	5	4

For SRTF, our schedule is (arrival on top):



Average waiting time: $\frac{9+1+0+2}{4} = 3$

So far we haven't considered fairness. We can make a scheduler more fair by using a round-robin scheduler, which is a pre-emptive scheduler which divides execution time into quanta and gives processes <quanta> of time while round-robining through them. [12]

[12] *How to consider quantum length? Consider context switching time*

Process	Arrival Time	Burst Time
P_1	0	7
P_2	2	4
P_3	4	1
P_4	5	4

For RR, our schedule is (arrival on top, queue on bottom):

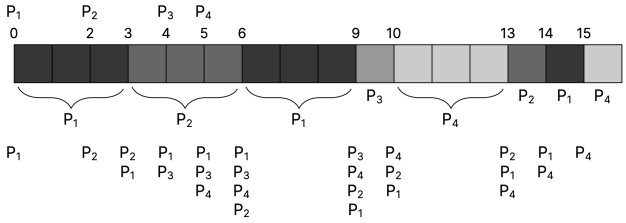


Figure 28. RR example with quanta of 3 units. Average number of switches is 7, average waiting time is $\frac{8+8+5+7}{4} = 7$, average response time is $\frac{0+1+5+5}{4}$. Note that ties are handled by favouring new processes.

Round robin performance is dependent on quantum length and job length. Long quantum causes starvation (FCFS), but twoo low and the performance sucks since context switches introduce overhead. If jobs have similar lengths RR has poor average waiting time