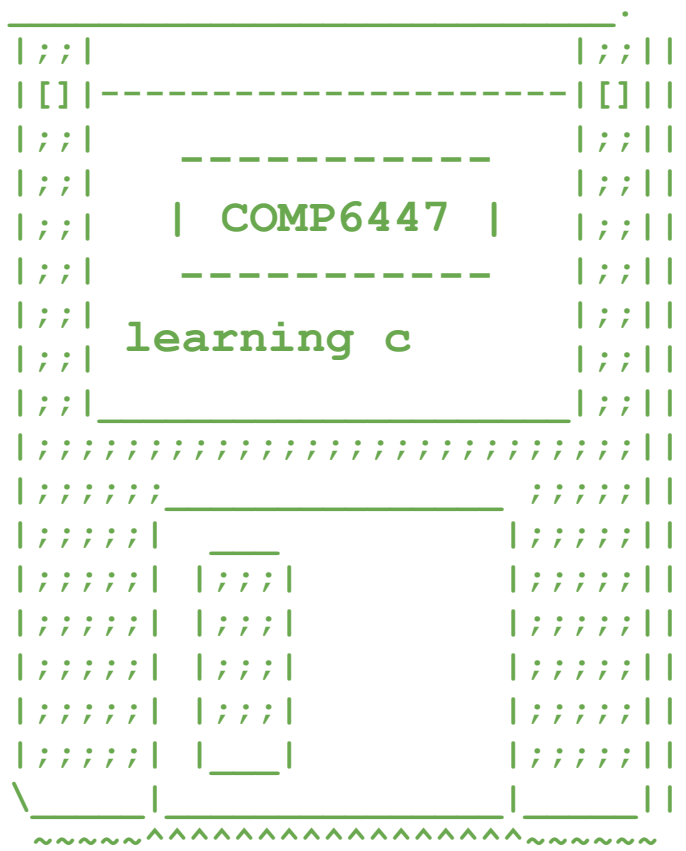

COMP6447
learning c

Source code auditing

And **fuzzzzzing**

Why do we read the source

1. Because it’s the best way to find bugs
2. Because it’s easier than reversing assembly
3. A lot of people prefer to fuzz and manipulate input
    a. They won’t find the bugs you will

Writing exploits is **easier** if you **understand** the **program**

Read the Source luke

Several classes of problems

- Bad api usage
- Logic bugs
- Integer overflows
- Type conversions / confusions
- Array out of bounds access
- Incorrect use of operators (sizeof / bit shifts / etc)
- Not understanding pointer arithmetic
- Race conditions

Let’s talk about **some** of these

Bad api usage

- Using apis in **stdlib** assume the **programmer** has a **brain**
  - Big mistake
- Not checking sizes
- Classic format string

```
Int main(int argc, char* argv[]) {

    printf(argv[1]);

}
```

Ordering parameters wrong...

strcpy?

```
Int main(int argc, char* argv[]) {
        Char s[4];
        strcpy(s, argv[1]);
}
```

More common than gets, less obvious to the developer why its bad

But im a good dev adam. I use **strncpy**

char *strncpy(char *dest, const char *src, size_t n);

- Doesn’t copy NULL byte if the size doesn’t allow it

Hmm...

Whats the problem here

```
int main(int argc, char* argv[]) {
    Char important[16] = “ABCDEFGHIKLMNO”;
    Char b[16];
    strncpy(b, argv[1], 16);
    printf(“%s\n”, b);
}
```

Do you think developers are good?

- The way C is compiled / typed, sometimes using an API a wrong way is a valid way to use it
- Look at memset
  - memset(void* s, int c, size_t n);
- Fills (**s**) with **n** bytes of **c**

Seems reasonable right?

Does this zero out memory?

- **memset(s, 100, 0)**

Integer overflows and underflows...

Unsigned int a;

inr a = INT_MAX

a += 1

What’s a?

unsigned int a = 0

a--;

What’s A?

```
u_char *    make_table(unsigned int width, unsigned int height,
u_char *init_row)
{
    unsigned int n;
    int i;
    u_char *buf;
    n = width * height;
    buf = (char *)malloc(n);

    if (!buf) return (NULL);

    for (i=0; i< height; i++)
        memcpy(&buf[i*width], init_row, width);
    return buf;
}
```

What happens if you multiply 2 big numbers

What happens when you multiply two massive numbers?

0x400 * 0x1000001 = 0x400000400 overflows to become 0x400 lol

Easy overflow.. Easy to spot

This is from an open source tool...

```
u_int nresp;
...
nresp = packet_get_int();
if (nresp > 0) {
    response = xmalloc(nresp * sizeof(char*));
    for (i = 0; i < nresp; i++)
            response[i] = packet_get_string(NULL);
    }
packet_check_eom();
```

What does sizeof(char*) return?

length+1 might overflow

```
char *read_data(int sockfd)
{
   char *buf;
   int length = network_get_int(sockfd);
   if(!(buf = (char *)malloc(MAXCHARS)))
      die("malloc: %m");
   if(length < 0 || length + 1 >= MAXCHARS){
      free(buf);
      die("bad length: %d", value);
   }
   if(read(sockfd, buf, length) <= 0){
      free(buf);
      die("read: %m");
   }
   buf[value] = '\0';
   return buf;
}
```

is this sudo vulnerable?

```
void assume_privs(unsigned short uid)
{
   seteuid(uid);
   setuid(uid);
}

int become_user(int uid)
{
   if (uid == 0)
      die("root isnt allowed");
   assume_privs(uid);
}
```

- Become_user takes an int
- Assume_priv takes a short
- An int like 0x1000 0000 would get truncated to 0x0000
  - The first check passes
  - But they still get root
  - Guess where i pulled this example from lol

What is a race condition?

A race condition is a **generic situation** where **two or more actors** are about to perform an action, and the result depends on the order in which they occur

example

- Checks file doesn’t exist
- Does some stuff
- Opens file
- Appends to file
- Saves file

```
#define DELAY 10000

int main() {
  char *fn = "/tmp/XYZ";
  char buffer[60];
  FILE *fp;

  long int i;
  /* get user input */
  scanf("%50s", buffer);
  if (!access(fn, W_OK)) {
    /* simulating delay */
    for (i = 0; i < DELAY; i++) {
      int a = i^2;
    }

    fp = fopen(fn, "a+");
    fwrite("\n", sizeof(char), 1, fp);
    fwrite(buffer, sizeof(char), strlen(buffer), fp);
    fclose(fp);
  }

  else
    printf("No permission \n");
```

example

- Checks file doesn't exist
- Does some stuff
- **Attacker creates symlink it to /etc/shadow**
- Opens file
- Appends to file
- Saves file

```
#define DELAY 10000

int main() {
  char *fn = "/tmp/XYZ";
  char buffer[60];
  FILE *fp;

  long int i;
  /* get user input */
  scanf("%50s", buffer);
  if (!access(fn, W_OK)) {
    /* simulating delay */
    for (i = 0; i < DELAY; i++) {
      int a = i^2;
    }

    fp = fopen(fn, "a+");
    fwrite("\n", sizeof(char), 1, fp);
    fwrite(buffer, sizeof(char), strlen(buffer), fp);
    fclose(fp);
  }

  else
    printf("No permission \n");
```

Source auditing is similar to reversing

- Look at the big picture
- Audit functions you know are unsafe
  - Printf, strcpy, gets
- Make a list of interesting functions as you audit
  - Try to understand what a function does by its name first
  - Don’t jump between functions too much, you’ll confuse yourself
- Bottom up approach
  - Start at user input
  - Move up and see where your data goes
- Read the manual


COMP6447
Fuzzing

Fuzzing

- Fuzzing or fuzz testing is an automated software testing technique that involves providing invalid, unexpected, or random data as inputs to a computer program.
- Not always used in a security sense, can be used for unit testing of software.

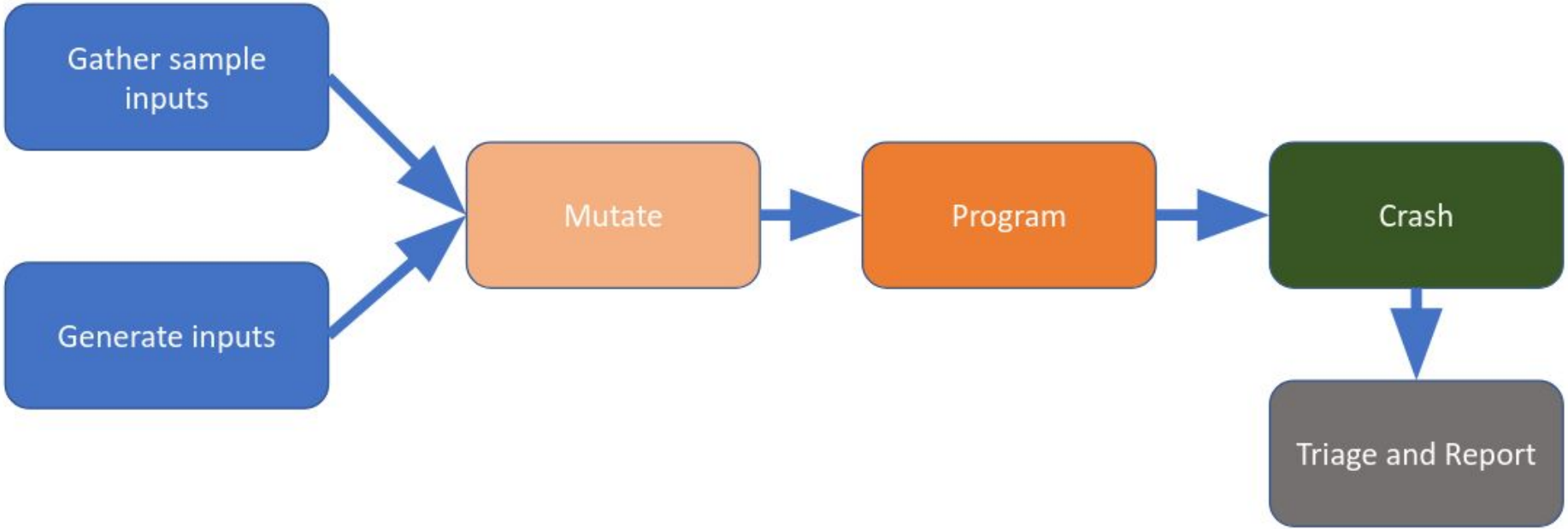
tldr;

- automate the finding of vulnerabilities
  - Generate test cases for program
  - Send them to program
  - Monitor for any changes in the programs behaviour
- Goal is to get the most code **coverage**
  - If you can test as many code **paths** as possible, more likely to find bugs
  - Monitoring code coverage is hard

More on fuzzing

- Fuzzing is often used to test file formats or networking protocols
  - Structured inputs
- Usually a fuzzer takes in a sample input, and mutates it
  - Or takes in **large sets** of inputs, and mutates them
- Goal is to create an exceptional condition
  - Program crashes
  - Hangs (infinite loops?)
  - Unique errors/error codes

Fuzzer **cycle**



Fuzzing has limitations compared to auditing

- Custom protocols / file formats are hard to know about without prior reversing
- Non crashing bugs are tough to detect
  - Arbritrary file read/write
  - SQL injection
  - Priv escs
- Hard to know what progress you have
  - Sometimes you will get no crashes for a long time.
    - Does this mean no bugs?
    - Is your fuzzer broken?
  - Sometimes you will get **a lot** of crashes
    - Maybe your fuzzer is triggering the same bug over and over
    - Not finding more useful bugs
- Most off the shelf fuzzers require source code for the program
- Generating all possible inputs to a program is not possible

Fuzzing has 2 main parts

- The fuzzer
- The harness

Harness

- The thing you use to watch the program behave
  - Could be a **debugger**
  - Could be **code coverage** stuff built into the binary
  - Can use a **hypervisor** to fuzz an entire system
  - Could just look at the **return code** after executing the program
- Stuff to look for
  - Crashes
  - Weird program states
  - Code coverage
  - Error messages
    - Information leaks?
- tldr: monitors for crashes, records what inputs caused what crash

How to start out writing your own fuzzer

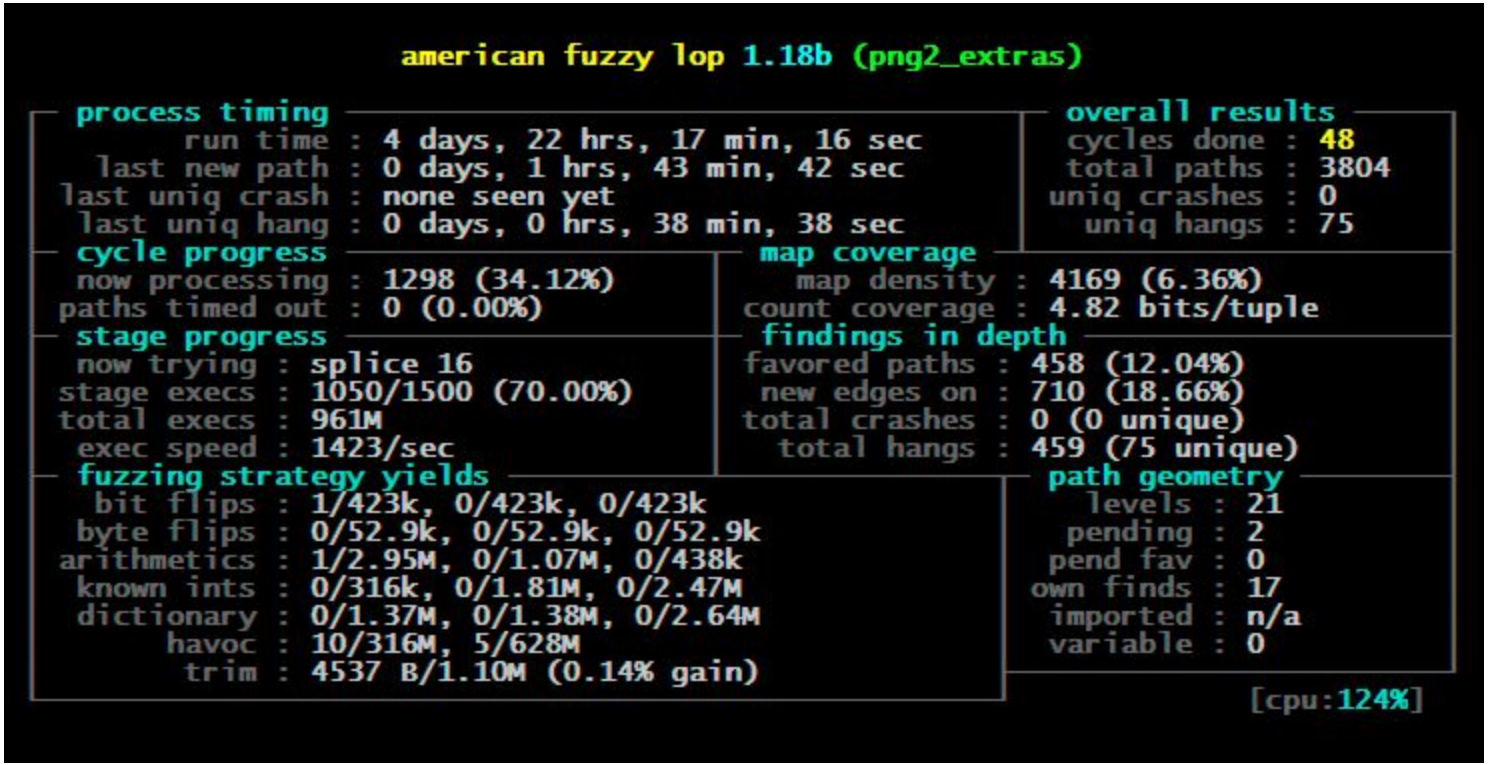
Start by just running the program

- Write a tool that takes some data, mutates it, and saves the mutated file
- Run the program to parse the mutated file
- Did the program crash?
  - Attach gdb or enable core dumps :D
  - Or look at return codes
- This has some big negatives
  - Can’t really scale, if the program relies on one file
  - Program startup is slow

Open source fuzzers

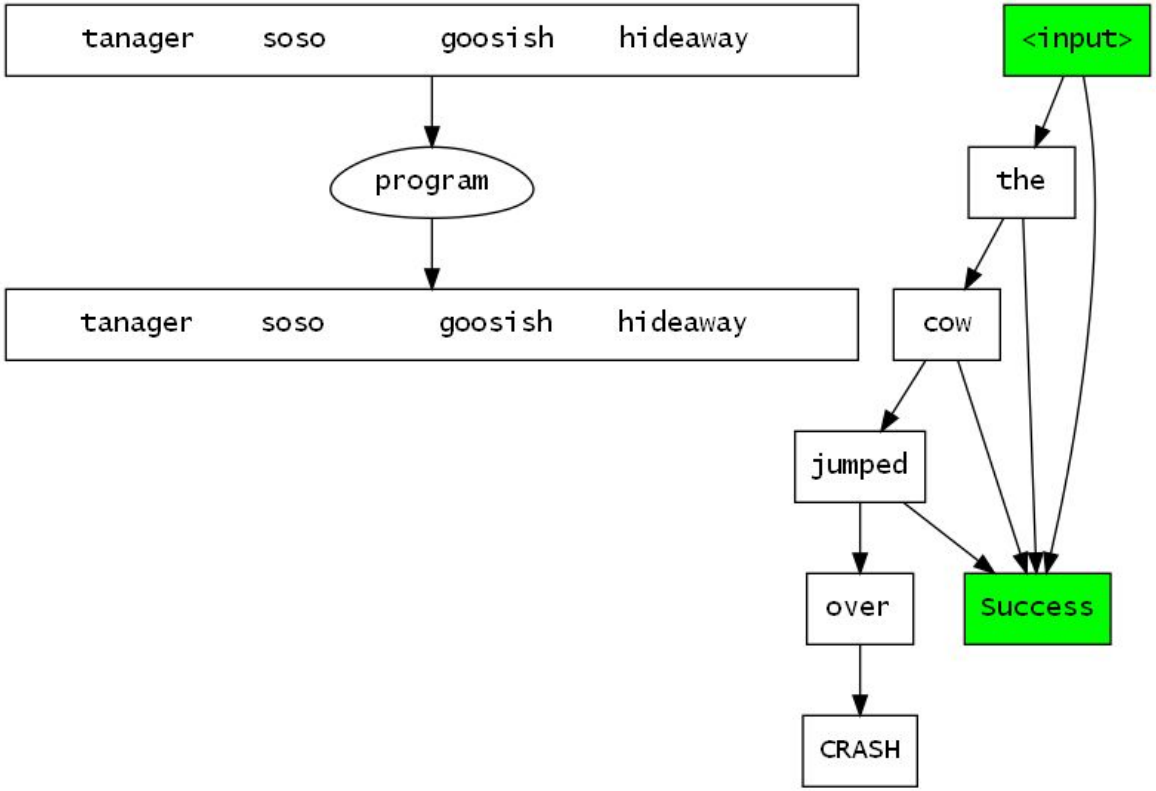
AFL

- One of the most used fuzzers
  - Libfuzzer is also good
- **Relies on source being available***
  - Looks at gaining most coverage
- Takes in sample tests (corpus based)
  - Mutates them
  - Looks at output
  - Keeps mutated files that reveals new code paths
  - Repeatedly mutate the file using a balanced and well-researched variety of traditional fuzzing strategies
- **Coverage guided fuzzing**


american fuzzy lop 1.18b (png2_extras)
process timing
run time : 4 days, 22 hrs, 17 min, 16 sec
last new path : 0 days, 1 hrs, 43 min, 42 sec
last uniq crash : none seen yet
last uniq hang : 0 days, 0 hrs, 38 min, 38 sec
overall results
cycles done : 48
total paths : 3804
uniq crashes : 0
uniq hangs : 75
cycle progress
now processing : 1298 (34.12%)
paths timed out : 0 (0.00%)
map coverage
map density : 4169 (6.36%)
count coverage : 4.82 bits/tuple
stage progress
now trying : splice 16
stage execs : 1050/1500 (70.00%)
total execs : 961M
exec speed : 1423/sec
findings in depth
favored paths : 458 (12.04%)
new edges on : 710 (18.66%)
total crashes : 0 (0 unique)
total hangs : 459 (75 unique)
fuzzing strategy yields
bit flips : 1/423k, 0/423k, 0/423k
byte flips : 0/52.9k, 0/52.9k, 0/52.9k
arithmetics : 1/2.95M, 0/1.07M, 0/438k
known ints : 0/316k, 0/1.81M, 0/2.47M
dictionary : 0/1.37M, 0/1.38M, 0/2.64M
havoc : 10/316M, 5/628M
trim : 4537 B/1.10M (0.14% gain)
path geometry
levels : 21
pending : 2
pend fav : 0
own finds : 17
imported : n/a
variable : 0
[cpu:124%]

Coverage based fuzzing

Follow code sections



Finding more important bugs

Not every bug causes a crash

- Small Overflows might not crash
- Integer overflows/underflows might not crash
- Invlaid Auth success might not crash

Add hooks into the program at these points, crash if condition is true.

This way you can find when these things happen

Address **San**atizer is one way of doing this, it crashes a program on invalid memory writes/reads

More open source

- Libfuzzer is another popular one
- Part of LLVM, designed to be built with your program
- Really fast because everything is done in memory (no files read from disk)
- Is corpus based similar to AFL

How to make a better fuzzer?

- Improve performance
  - Moar threads
    - Not too many though, don’t kill the machine :D
- Most fuzzers are slow
- Relying on disk + syscalls is slow
  - So don’t?
  - Ie: Pass files through memory

Example fuzzer input/out

Sample input:

- GET /index.html HTTP/1.1

Fuzzer output:

- AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
- AAAAAAAAAA….AAAAAAAAAA /index.html HTTP/1.1
- GET %n%n%n%n%n%n%n HTTP/1.1
- GET /index.html HTTTTTTTTTP/1.1
- GET /index.html HTTP/0.0
- etcetcetc

Example vulnerable code

```
char buffer[1024]

strcpy(buffer, argv[1])

return 0;
```

```
while True:
    subprocess.call([“./program”, “A” * i])
    if crash:
        print(‘we got em”)
```

Fuzzing is about **intelligently** generating input...

Fuzzing strategies

Mutation based fuzzing - Dumb but easy

Generation based fuzzing - Smart but hard

Mutation based fuzzing

- No understanding of structure of input
- Completely random inputs
- Add completely random anomalies to existing input
- Easy and quick to setup

Generation based fuzzing

- Generate input based on some format / test case
  - Example: input must follow some spec, etc
- Anomalies are added at each spot in the input
  - If json, field names are fuzzed, field values are fuzzed, arrays are fuzzed
  - If xml, ……
  - Situational, you must know what the data looks like
- Knowledge of protocol gives a lot more insight compared to random fuzzing

Example fuzzer demo

- Lets hope this demo of a bitflipper works

What do you do if you keep triggering the same bug..

- Add tests to your fuzzer to ignore it
- Try a different technique (never stick to one technique)
- Given a crash.. How do you find the actual vulnerability and exploit it
- When do you give up on fuzzing

Assignment...

Write a fuzzer in groups of 4.

Your goal is to **write a fuzzer** that **given a sample input and a binary** will **generate a file** that when passed into the **program**, **causes a crash.**

Will be given 10 sample binaries + 10 sample inputs.

Will be tested on **more.**

Assignment is intentionally designed with a wide scope. Be creative...

Example test case

```
$ ls
fuzzer.exe input.txt binary
$ ./fuzzer.exe binary input.txt
Fuzzing this thing...
$ ls
fuzzer.exe binary bad.txt input.txt
$ cat bad.txt | binary
Segmentation Fault
```

Marks (30 max marks)

20 marks - for finding all vulnerabilities in the **10 provided binaries**.

10 marks - for finding all vulnerabilities in the **binaries that we do not provide**.

6 marks - Something awesome . Something **cool** your fuzzer does.

Some hints / assumptions

Writing a fully black box fuzzer is hard. You can assume some things about the input:

- Input will be one of the following text formats:
  - Plaintext (multiline)
  - yaml
  - json
  - xml
  - csv
- Bugs might be in the program or the parser :-)