

Midterm wargame

Wed-Fri - 3 questions

- 2 hours
- 1 buffer overflow
- 2 fmt string



<https://midterm.6447.lol/>

Wed 9am (after lecture) - Fri 6pm. (Must submit **by 6pm**)

Next week flex week

- No new content
- No tutorials
- **Optional** - Will be walking through midterm solutions and how to solve/approach the challenges in exam conditions on normal Monday Lecture slot

[illegible]

Source code auditing

And **fuzzzzzzing**



Why do we read the source

- Because it is the best way to find the best bugs
- You need to do a lot of this before you can start doing the same thing in a reverse context
- So many people are resistant to actually reading the source; love to spend time playing with the program by manipulating input, no desire to understand.
- Mark Dowd, “I'll never stop to be amazed by the amount of efforts people put in not understand things”
- When writing exploits, having a precise understanding of **what is going on is super important**

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...

gets?

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


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...

Is strncpy safe?

```
char *strcpy(char *dest, const char *src)
```

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

DESCRIPTION

The `strcpy()` function copies the string pointed to by `src`, including the terminating null byte (`'\0'`), to the buffer pointed to by `dest`. The strings may not overlap, and the destination string `dest` must be large enough to receive the copy. Beware of buffer overruns! (See BUGS.)

The `strncpy()` function is similar, except that at most `n` bytes of `src` are copied. Warning: If there is no null byte among the first `n` bytes of `src`, the string placed in `dest` will not be null-terminated.

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)`**

Is anyone really this dumb



Results 1 - 10 of 160 (0.881 seconds)

<http://mtasa-resources.googlecode.com/svn> - Unknown - C

linux-2.4/arch/s390/kernel/smp.c

[⊕ Show duplicates](#)

```
529:    reschedule the forked task. */
530: memset(&regs,sizeof(pt_regs),0);
531: return do_fork(CLONE_VM|CLONE_PID, 0, &regs, 0);
```

<http://www.netwinder.org/mirror/users/t/tpoole/linux-2.4.tgz> - GPL - C

trunk/core/x86/loadtoconst.c

```
91: memset(regs_modified,sizeof(bool)*8,0);
```

<http://dynamorio.googlecode.com/svn> - Unknown - C

mtools-3.9.10/fat.c

```
449: set_dword(infoSector->signature1, INFOSECT_SIGNATURE1);
450: memset(infoSector->filler1, sizeof(infoSector->filler1),0);
451: memset(infoSector->filler2, sizeof(infoSector->filler2),0);
452: set_dword(infoSector->signature2, INFOSECT_SIGNATURE2);
```

<ftp://ftp.usa.openbsd.org/pub/OpenBSD/distfiles//mtools-3.9.10.tar.gz> - GPL - C

Heap

Heap overflows are a huge issue.

Although lots of these are standard overflows as per *cpy or in a loop, some are related to specifics of heap implementation.

le: Use After Free

```
{  
    char *s = malloc(n);  
    free(malloc);  
    function(s);  
}
```

Can be thousands of lines apart..

NULL pointers and malloc

Not checking if malloc() worked is bad!

malloc() will return NULL, which almost everywhere is 0. Then we'll try to access the page at address 0x0! With some tricks, **this can be exploitable**. Most of the time, just a DoS

Double frees() can be exploitable, but they're somewhat trixy and depends on the heap library:

```
{  
    char *s=malloc();  
    free(s);  
    free(s);  
}
```


Type conversions

You can convert between types explicitly and automatically, may produce a warning:

Explicit conversion: `if ((int) c == i)`

Automatic conversion: `if (c == i) printf("%d\n", p); //-1262761449`

Integer overflows and underflows...

What's a?

```
int a = INT_MAX
```

```
a += 1
```

```
unsigned int a = 0
```

```
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

openssh 3.1

```
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 about Jim and Bob

```
int main(void)
{
    unsigned char jim = 255;
    unsigned char bob = 255;
    if ((jim + bob) > 300)
        printf("yes\n");
    else
        printf("no\n");
}
```

What about Jim and Bob

Why is this **Yes**

```
int main(void)
{
    unsigned char jim = 255;
    unsigned char bob = 255;
    if ((jim + bob) > 300)
        printf("yes\n");
    else
        printf("no\n");
}
```

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;

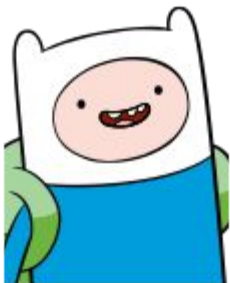
    long int i;
    /* get user input */
    scanf("%50s", buffer);
    if (!access(fn, W_OK)) {
        /* simulating delay */
        for (i = 0; i < DELAY; i++) {
            int a = i^2;
        }

        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

[illegible]

Fuzzing

- Fuzzing is a testing technique used to **discover vulnerabilities** in software by **inputting random or malformed data**.
- 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

Types of fuzzers

Black-box Fuzzing: No knowledge of the internal workings of the application.

White-box Fuzzing: Uses knowledge of the source code to generate inputs.

Grey-box Fuzzing: Combines both approaches, using some insights into the code without full access.

Fuzzing Techniques

Mutation-based Fuzzing: Modifies existing inputs to create new test cases.

Generation-based Fuzzing: Creates inputs from scratch based on a specification.

Grammar-based Fuzzing: Uses a formal grammar to generate valid inputs.

History of fuzzin

- Barton Miller at University of Wisconsin "Operating System Utility Program Reliability -The Fuzz Generator" -1989
- Dave Aitel released two fuzzers in 2001 -SPIKE fuzzing framework, and sharefuzz. Spike in particular was the first block based fuzzer publicly known, and made fuzzing a huge thing in the security world –2001
- Big gap there of nothing..
- Many fuzzing toolkits have been released since then

Why fuzz

- Reappliable effort –Fuzzer creation is generally focused on protocol, file format, etc. –so you can take your PDF fuzzer and use it on every PDF parser, or take your SSL fuzzer and use it on every SSL implementation
- Write once, use many times
- Lower skill requirement than in-depth reverse engineering or source code review
- Bugs found are guaranteed to be reachable, if you are fuzzing from reachable interface
- Largely, better bang for buck

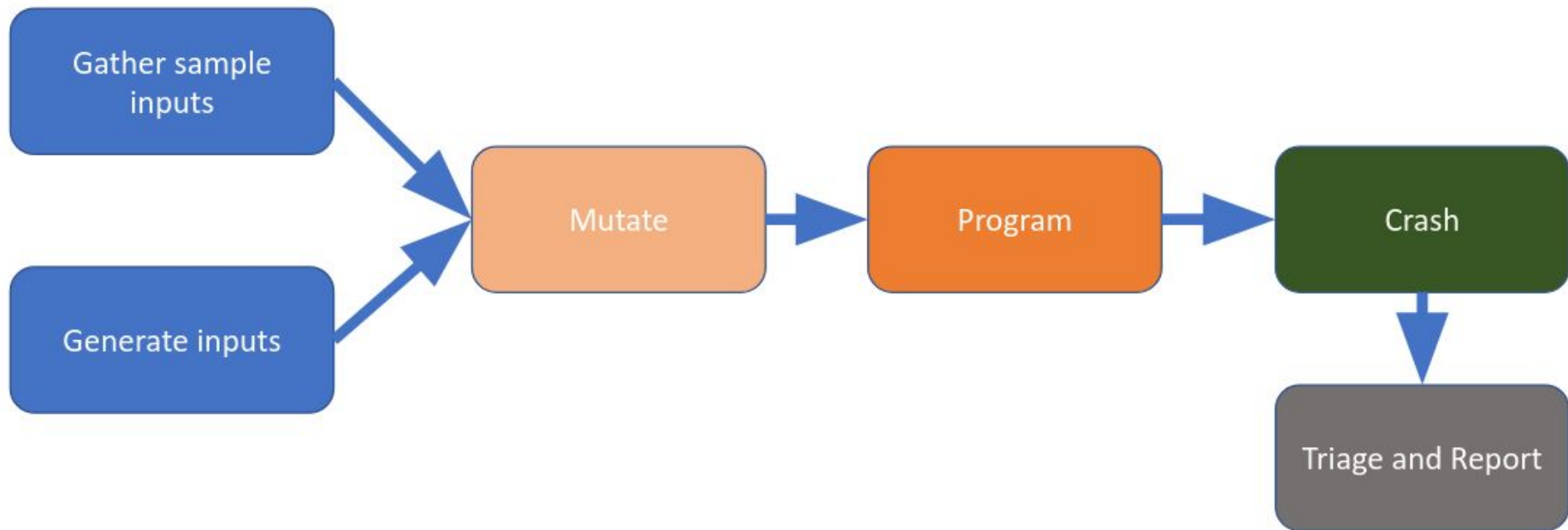
Cons of fuzzing

- Bugs are often shallow -Other people may find them. Now that exploit writing time is often very large (compared to vuln finding time)
- Hard to get full coverage of program –remember that the least used paths are more likely to be vulnerable
- Multistage vulns—i.e. those which require multiple concurrent things to happen to occur are hard to target –i.e. think about a network protocol, where if the first packet of a particular type of communication is slightly malformed in a certain way, and then the second packet is in a different way, then a bug occurs. Very hard to cover (but not impossible..)
- Will not find smarter things like backdoors or complex race conditions

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
 - Arbitrary 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
 - Creates test cases via generation or mutation
- The harness
 - Monitor process
 - Starts application
 - Feeds it input
 - Rinse and repeat
 - Gather statistics / diagnostics

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
 - Or dmesg, etc
- 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**

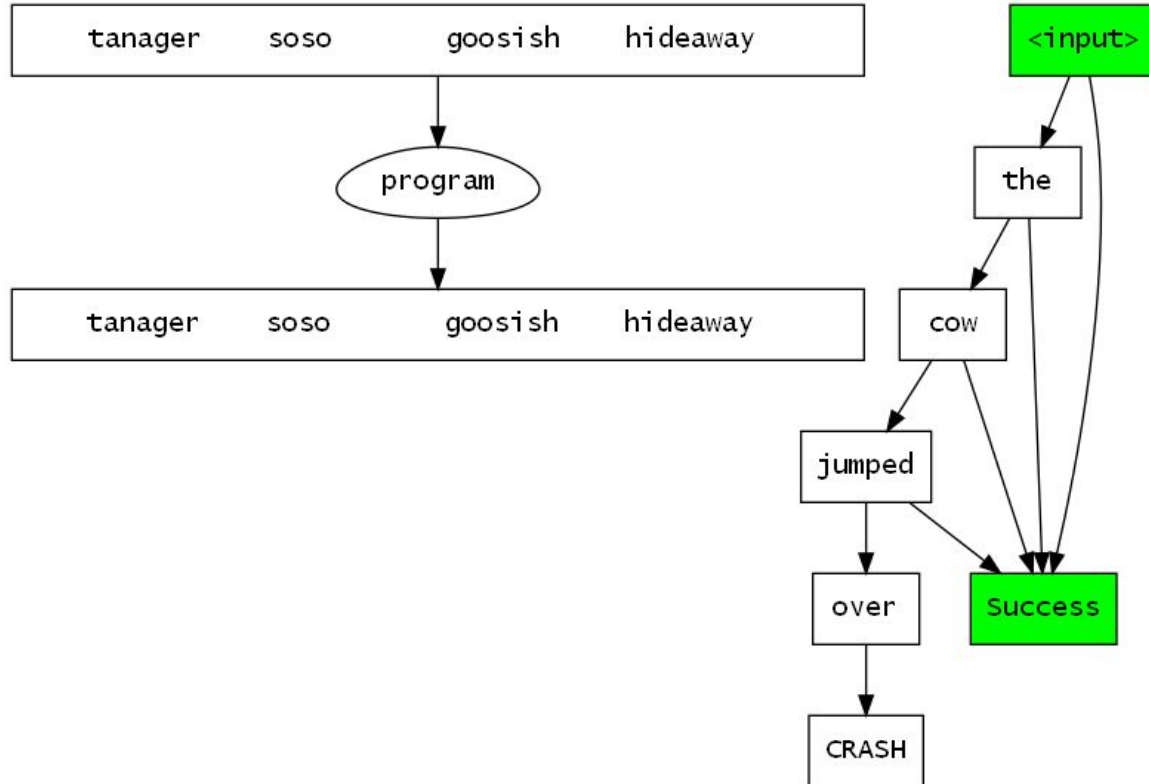
american fuzzy lop 1.18b (png2_extras)

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

[cpu:124%]

Coverage based fuzzing

Follow code sections



Hardest to reach code is often
the buggiest..

Finding more important bugs

Not every bug causes a crash

- Small Overflows might not crash
- Integer overflows/underflows might not crash
- Invalid 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 Sanitizer 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:

- AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
- AAAAAAAAAA....AAAAAAAAAA /index.html HTTP/1.1
- GET %n%n%n%n%n%n HTTP/1.1
- GET /index.html HTTTTTTTTTTP/1.1
- GET /index.html HTTP/0.0
- etcetcetc

Example vulnerable code

```
char buffer[1024]
```

```
strcpy(buffer, argv[1])
```

```
return 0;
```

```
for i in range(1, infinity):
```

```
    subprocess.call(["./program", "A" * i])
```

```
    if crash:
```

```
        print('we got em')
```

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

Fuzzing strategies

```
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)
```

Mutation based fuzzing - Dumb but easy

Generation based fuzzing - Smart but hard

Grammar-based Fuzzing (Can also use subset of above)

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

```
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)
```

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

fuzzers are useful...

Ilja's 5 line mangler python script found a lot of bugs

FROM ILJA'S TALK (CIRCA 2005) WHAT HIS MANGLER FOUND

- libmagic (used file)
- preview (osX pdf viewer)
- xpdf (hang, not a crash ...)
- mach-o loading
- qnx elf loader (panics almost instantly, yikes !)
- FreeBSD elf loading
- openoffice
- amp
- osX image loading (.dmg)
- libbfd (used objdump)
- libtiff (used tiff2pdf)
- xine
- OpenBSD elf loading (3.7 on a sparc)
- unixware 713 elf loading
- DragonFlyBSD elf loading
- solaris 10 elf loading
- cistron-radiusd
- linux ext2fs (2.4.29) image loading
- linux reiserfs (2.4.29) image loading
- linux jfs (2.4.29) image loading
- linux xfs (2.4.29) image loading
- macromedia flash parsing
- Totem 0.99.15.1
- Gnumeric
- Quicktime
- Mplayer
- Python byte interpreter
- Realplayer (10.0.6.776)
- Dvips
- Php 5.1.1
- IE 6
- OS X WebKit (used safari)

A few years back...

- One of the students wrote a DHCP fuzzer
- Tested dhcpcd-3, dhcpcd-3, Dual DHCP-DNS server and Tiny DHCP Server
- Managed to find a buffer overflow in dual dhcp-DNS
 - the server trusted a length field from the client

Corporate worlds have fuzzers

- Big name companies have started incorporating fuzzing in their normal testing suite
- Microsoft has several main fuzzers
- Adobe have some fuzzers but they must suck...

OMG. It looks like you're developing a pdf reader with your own parser.
Would you like to read the fucking news?

9:40 AM Apr 6th via Seismic



DevPaperclip
Developer Paperclip

Final fuzzing slide

- Even dumb fuzzing is super powerful
- As future developers/hackers you should know about it and use it as much as possible
- Often a battle of CPU not of brains
- Amazon EC2 can make it interesting
 - \$5 per hour for a windows server with 16 GB of memory, 20 EC2 Compute Units (8 virtual cores), 1690 GB of local instance storage, 64-bit platform

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.

Will be tested on **more**.

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

Assignment is ambiguous and mostly self taught

Good in interviews so put effort into it :)

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
```

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
 - ...others
- Bugs might be in the program or the parser :-)

Spec

Spec: fuzzer.6447.lol