

Secure Software Development

Finding Bugs II

Daniel Gruss, Vedad Hadzic, Andreas Kogler, Martin Schwarzl, Marcel Nageler

12.11.2021

Winter 2021/22, www.iaik.tugraz.at

1. Fuzzing
2. Symbolic Execution
3. Memory Debugging
4. Reverse Engineering
5. Binary Diffing

PREVIOUSLY ON

SSD

- **Human experts** can find bugs by looking at source code

- **Human experts** can find bugs by looking at source code
- **Static code analysis** finds many bugs by looking at source code

- **Human experts** can find bugs by looking at source code
- **Static code analysis** finds many bugs by looking at source code
- **Sanitizers** add bug-finding code to the source code

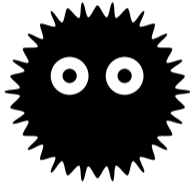
- **Human experts** can find bugs by looking at source code
- **Static code analysis** finds many bugs by looking at source code
- **Sanitizers** add bug-finding code to the source code
- **git bisect** can assist in finding bugs in source code

- **Human experts** can find bugs by looking at source code
- **Static code analysis** finds many bugs by looking at source code
- **Sanitizers** add bug-finding code to the source code
- **git bisect** can assist in finding bugs in source code
- **Explaining** source code helps in finding bugs

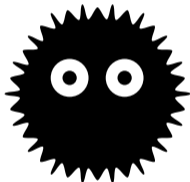
- **Human experts** can find bugs by looking at source code
- **Static code analysis** finds many bugs by looking at source code
- **Sanitizers** add bug-finding code to the source code
- **git bisect** can assist in finding bugs in source code
- **Explaining** source code helps in finding bugs
- Adding **print statements** to source code can assist in finding bugs

- **Human experts** can find bugs by looking at source code
- **Static code analysis** finds many bugs by looking at source code
- **Sanitizers** add bug-finding code to the source code
- **git bisect** can assist in finding bugs in source code
- **Explaining** source code helps in finding bugs
- Adding **print statements** to source code can assist in finding bugs
- Stepping through source code with a **debugger** often reveals bugs

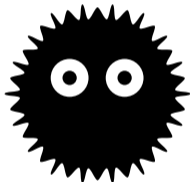




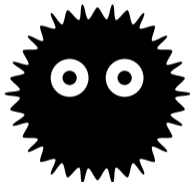
- **Fuzzing** is an automated software testing method



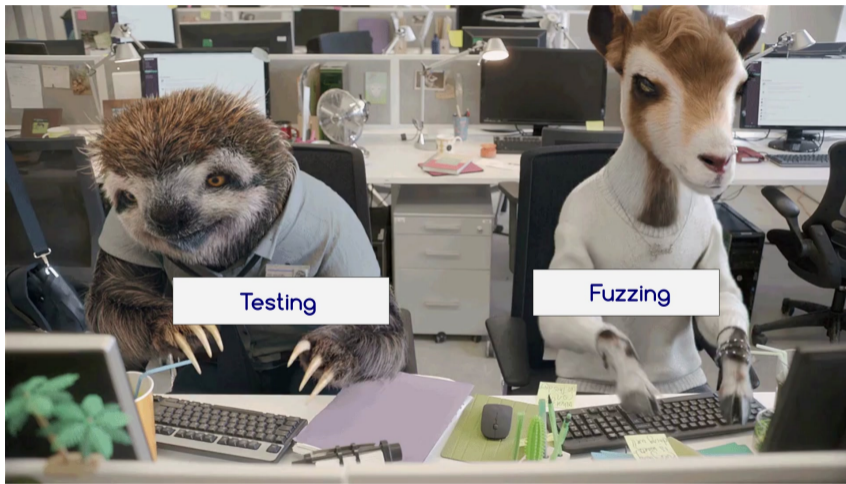
- **Fuzzing** is an automated software testing method
- Provide invalid, unexpected, or random data as input



- **Fuzzing** is an automated software testing method
- Provide invalid, unexpected, or random data as input
- Monitor program behavior (e.g., crash, assertion, ...)



- **Fuzzing** is an automated software testing method
- Provide invalid, unexpected, or random data as input
- Monitor program behavior (e.g., crash, assertion, ...)
- Typically used for **file formats** or protocols



Fuzzing

Testing

Fuzzing

- **Invalid**, unexpected, or random data as input

Testing

- **Normal**, valid, well-formed data as input

Fuzzing

- **Invalid**, unexpected, or random data as input
- **Automatically** generated testcases

Testing

- **Normal**, valid, well-formed data as input
- **Manually** generated testcases

Fuzzing

- **Invalid**, unexpected, or random data as input
- **Automatically** generated testcases
- **Fun** 😊

Testing

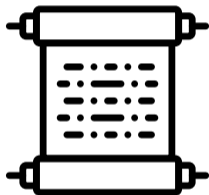
- **Normal**, valid, well-formed data as input
- **Manually** generated testcases
- **Boring** 😞

Fuzzing

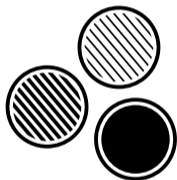
- **Invalid**, unexpected, or random data as input
- **Automatically** generated testcases
- **Fun** 😊
- **Goal:** Find exploitable errors

Testing

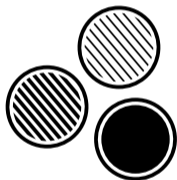
- **Normal**, valid, well-formed data as input
- **Manually** generated testcases
- **Boring** 😞
- **Goal:** Normal users don't get errors



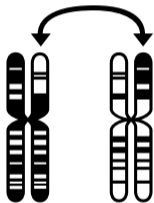
- Fuzzing dates back to 1981, was considered worst means of testing
- Term was coined 1988: testing noise over fuzzy network connections
- Google runs **ClusterFuzz** since 2012 to fuzz Chrome
- Most teams used fuzzing to automatically detect bugs in the DARPA **Cyber Grand Challenge** 2016
- **american fuzzy lop** (AFL) regularly finds bugs in open source programs



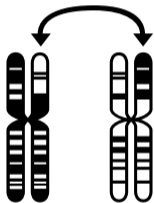
- Fuzzing can be somewhere between **dumb** and **smart**
- The smarter the fuzzing, the harder the setup



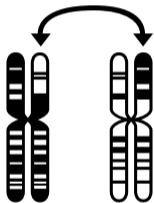
- Fuzzing can be somewhere between **dumb** and **smart**
- The smarter the fuzzing, the harder the setup
- However, **smarter fuzzing** finds **more bugs**



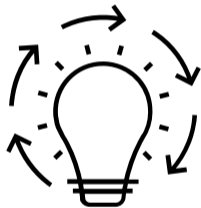
- Mutate existing data sets to generate new testcases
- Mutation might be completely random or follow some pattern



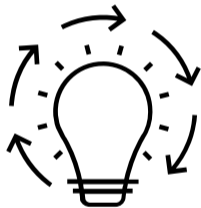
- Mutate existing data sets to generate new testcases
- Mutation might be completely random or follow some pattern
- + No knowledge of the input structure is required
- + Easy to set up



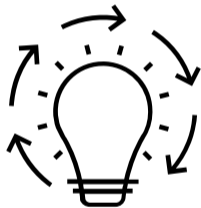
- Mutate existing data sets to generate new testcases
- Mutation might be completely random or follow some pattern
- + No knowledge of the input structure is required
- + Easy to set up
- Might fail for complicated protocols (e.g., challenge response)
- Fails for complex file formats (e.g., checksum)



- Testcases are generated from a format description (e.g., RFC)
- Anomalies are added to each field of the input



- Testcases are generated from a format description (e.g., RFC)
- Anomalies are added to each field of the input
- + Knowledge of protocol gives better results
- + More targeted, *i.e.*, does not fuzz “uninteresting” data
- + Handles complex dependencies, e.g., checksums



- Testcases are generated from a format description (e.g., RFC)
- Anomalies are added to each field of the input
- + Knowledge of protocol gives better results
- + More targeted, *i.e.*, does not fuzz “uninteresting” data
- + Handles complex dependencies, e.g., checksums
- Requires specification of protocol
- Writing testcase generator is a lot of work

- Generate inputs/mutations based on program response
- Different metrics: code coverage, reaching potentially dangerous functions, ...





- Generate inputs/mutations based on program response
- Different metrics: code coverage, reaching potentially dangerous functions, ...
- + Dynamically learns protocols, no configuration needed
- + Finds many bugs
- + Robust and fast



- Generate inputs/mutations based on program response
- Different metrics: code coverage, reaching potentially dangerous functions, ...
- + Dynamically learns protocols, no configuration needed
- + Finds many bugs
- + Robust and fast
- Does not handle large input files well
- Often requires binary instrumentation



Practical Example: Fuzzing



```
typedef void (*function)(char*);

typedef struct {
    char* name;
    function func;
} functions;

int is_valid(functions* list, char* name) {
    int i = -1;
    while(list[++i].name) {
        if(!strncmp(list[i].name, name, strlen(list[i].name))) return 1;
    }
    return 0;
}

void execute(functions* list, char* name, char* cmd) {
    int i = 0;
    function func;
    while(list[++i].name) {
        if(!strncmp(list[i].name, name, strlen(name))) {
            func = list[i].func;
            break;
        }
    }
    func(cmd);
}

void ping(char* cmd) {
    printf("Pong\n");
}
```

```
void pong(char* cmd) {
    printf("Ping\n");
}

void echo(char* cmd) {
    printf("%s", cmd);
}

int main(int argc, char **argv) {
    char buffer[64];
    functions list[] = {
        {"pong", pong},
        {"ping", ping},
        {"echo", echo},
        {NULL, NULL}
    };

    FILE* f = fopen(argv[1], "r");
    if(!f) return 1;
    while(fgets(buffer, 64, f)) {
        char* cmd = strtok(buffer, " \n");
        if(cmd) {
            if(is_valid(list, cmd)) {
                execute(list, cmd, strtok(NULL, ""));
            } else {
                printf("Unknown command!\n");
            }
        }
    }
    fclose(f);
    return 0;
}
```



```
% AFL_USE_ASAN=1 afl-gcc fuzz.c  
afl-cc 2.51b by <lcamtuf@google.com>  
afl-as 2.51b by <lcamtuf@google.com>  
[+] Instrumented 14 locations (64-bit, ASAN/MSAN mode, ratio 33%).
```



```
% AFL_USE_ASAN=1 afl-gcc fuzz.c
afl-cc 2.51b by <lcamtuf@google.com>
afl-as 2.51b by <lcamtuf@google.com>
[+] Instrumented 14 locations (64-bit, ASAN/MSAN mode, ratio 33%).
```

```
% cat afl-in/small
echo hi
ping
```



```
% AFL_USE_ASAN=1 afl-gcc fuzz.c
afl-cc 2.51b by <lcamtuf@google.com>
afl-as 2.51b by <lcamtuf@google.com>
[+] Instrumented 14 locations (64-bit, ASAN/MSAN mode, ratio 33%).
```

```
% cat afl-in/small
echo hi
ping
```

```
% afl-fuzz -m none -i afl-in -o afl-out -- ./a.out @@
afl-fuzz 2.51b by <lcamtuf@google.com>
[+] You have 4 CPU cores and 1 runnable tasks (utilization: 25%).
[...]
[+] All set and ready to roll!
```



```

american fuzzy lop 2.51b (a.out)

process timing |-----| overall results
  run time : 0 days, 0 hrs, 0 min, 2 sec | cycles done : 0
  last new path : 0 days, 0 hrs, 0 min, 1 sec | total paths : 7
  last uniq crash : 0 days, 0 hrs, 0 min, 2 sec | uniq crashes : 2
  last uniq hang : none seen yet | uniq hangs : 0
-----|-----|
cycle progress |-----| map coverage
  now processing : 0 (0.00%) | map density : 0.02% / 0.04%
  paths timed out : 0 (0.00%) | count coverage : 1.17 bits/tuple
-----|-----|
stage progress |-----| findings in depth
  now trying : interest 32/8 | favored paths : 1 (14.29%)
  stage execs : 250/431 (58.00%) | new edges on : 7 (100.00%)
  total execs : 1720 | total crashes : 247 (2 unique)
  exec speed : 541.6/sec | total tmouts : 0 (0 unique)
-----|-----|
fuzzing strategy yields |-----| path geometry
  bit flips : 4/96, 0/95, 0/93 | levels : 2
  byte flips : 0/12, 0/11, 0/9 | pending : 7
  arithmetics : 2/669, 0/55, 0/0 | pend fav : 1
  known ints : 2/64, 0/306, 0/0 | own finds : 6
  dictionary : 0/0, 0/0, 0/0 | imported : n/a
  havoc : 0/0, 0/0 | stability : 100.00%
  trim : 7.69%/3, 0.00%
-----|-----|
^C |-----|
                                     [cpu000: 81%]

+++ Testing aborted by user +++
[+] We're done here. Have a nice day!
    
```



```
% cat afl-out/crashes/id*  
pingJecho Hi
```




```
% cat afl-out/crashes/id*  
pingJecho Hi
```

```
% gdb --args ./fuzzing afl-out/crashes/id:000000  
(gdb) r  
Starting program: fuzzing afl-out/crashes/id:000000  
  
Program received signal SIGSEGV, Segmentation fault.  
0x0000000000000000 in ?? ()  
(gdb) bt  
#0 0x0000000000000000 in ?? ()  
#1 0x00000000004008ba in execute (list=0x7fffffffdb60,  
    name=0x7fffffffdba0 "pingJecho", cmd=0x7fffffffdbaa "Hi") at fuzzing.c:30  
#2 0x0000000000400a25 in main (argc=2, argv=0x7fffffffcd8) at fuzzing.c:60
```



The bug must be somewhere in `is_valid` and/or `execute`...

```
int is_valid(functions* list, char* name) {
    int i = -1;
    while(list[++i].name) {
        if(!strncmp(list[i].name, name,
                    strlen(list[i].name))) return 1;
    }
    return 0;
}
```

```
void execute(functions* list, char*
            name, char* cmd) {
    int i = 0;
    function func;
    while(list[++i].name) {
        if(!strncmp(list[i].name, name,
                    strlen(name))) {
            func = list[i].func;
            break;
        }
    }
    func(cmd);
}
```



The bug must be somewhere in `is_valid` and/or `execute`...

```
int is_valid(functions* list, char* name) {
    int i = -1;
    while(list[++i].name) {
        if(!strncmp(list[i].name, name,
                    strlen(list[i].name))) return 1;
    }
    return 0;
}
```

```
void execute(functions* list, char*
            name, char* cmd) {
    int i = 0;
    function func;
    while(list[++i].name) {
        if(!strncmp(list[i].name, name,
                    strlen(name))) {
            func = list[i].func;
            break;
        }
    }
    func(cmd);
}
```



The bug must be somewhere in `is_valid` and/or `execute`...

```
int is_valid(functions* list, char* name) {
    int i = -1;
    while(list[++i].name) {
        if(!strncmp(list[i].name, name,
                    strlen(list[i].name))) return 1;
    }
    return 0;
}
```

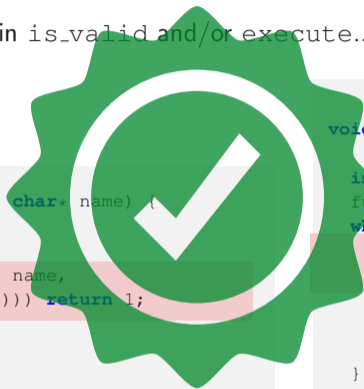
```
void execute(functions* list, char*
             name, char* cmd) {
    int i = 0;
    function func;
    while(list[++i].name) {
        if(!strncmp(list[i].name, name,
                    strlen(list[i].name))) {
            func = list[i].func;
            break;
        }
    }
    func(cmd);
}
```



The bug must be somewhere in `is_valid` and/or `execute`...

```
int is_valid(functions* list, char* name) {
    int i = -1;
    while(list[++i].name) {
        if(!strncmp(list[i].name, name,
                    strlen(list[i].name))) return 1;
    }
    return 0;
}
```

```
void execute(functions* list, char*
             name, char* cmd) {
    int i = 0;
    function func;
    while(list[++i].name) {
        if(!strncmp(list[i].name, name,
                    strlen(list[i].name))) {
            func = list[i].func;
            break;
        }
    }
    func(cmd);
}
```





```
% AFL_USE_ASAN=1 afl-gcc fuzz.c  
afl-cc 2.51b by <lcamtuf@google.com>  
afl-as 2.51b by <lcamtuf@google.com>  
[+] Instrumented 18 locations (64-bit, ASAN/MSAN mode, ratio 33%).
```



```
% AFL_USE_ASAN=1 afl-gcc fuzz.c  
afl-cc 2.51b by <lcamtuf@google.com>  
afl-as 2.51b by <lcamtuf@google.com>  
[+] Instrumented 18 locations (64-bit, ASAN/MSAN mode, ratio 33%).
```

```
% cat afl-in/small  
echo hi  
ping
```



```
% AFL_USE_ASAN=1 afl-gcc fuzz.c
afl-cc 2.51b by <lcamtuf@google.com>
afl-as 2.51b by <lcamtuf@google.com>
[+] Instrumented 18 locations (64-bit, ASAN/MSAN mode, ratio 33%).
```

```
% cat afl-in/small
echo hi
ping
```

```
% afl-fuzz -m none -i afl-in -o afl-out -- ./a.out @@
afl-fuzz 2.51b by <lcamtuf@google.com>
[+] You have 4 CPU cores and 1 runnable tasks (utilization: 25%).
[...]
[+] All set and ready to roll!
```




american fuzzy lop 2.51b (a.out)

```

process timing
  run time : 0 days, 0 hrs, 0 min, 2 sec
  last new path : 0 days, 0 hrs, 0 min, 0 sec
  last uniq crash : 0 days, 0 hrs, 0 min, 1 sec
  last uniq hang : none seen yet
cycle progress
  now processing : 0 (0.00%)
  paths timed out : 0 (0.00%)
stage progress
  now trying : arith 8/8
  stage execs : 456/718 (63.51%)
  total execs : 816
  exec speed : 311.4/sec
fuzzing strategy yields
  bit flips : 3/96, 1/95, 0/93
  byte flips : 0/12, 0/11, 0/9
  arithmetics : 0/0, 0/0, 0/0
  known ints : 0/0, 0/0, 0/0
  dictionary : 0/0, 0/0, 0/0
  havoc : 0/0, 0/0
  trim : 7.69%/3, 0.00%
overall results
  cycles done : 0
  total paths : 5
  uniq crashes : 1
  uniq hangs : 0
map coverage
  map density : 0.02% / 0.03%
  count coverage : 1.37 bits/tuple
findings in depth
  favored paths : 1 (20.00%)
  new edges on : 4 (80.00%)
  total crashes : 1 (1 unique)
  total tmouts : 0 (0 unique)
path geometry
  levels : 2
  pending : 5
  pend fav : 1
  own finds : 4
  imported : n/a
  stability : 100.00%
[C] [cpu000: 51%]

```

```

+++ Testing aborted by user +++
[+] We're done here. Have a nice day!

```



```
% cat afl-out/crashes/id*  
echo Hi  
pong
```



```
% cat afl-out/crashes/id*  
echo Hi  
pong
```

```
% gdb --args ./fuzzing afl-out/crashes/id:000000  
(gdb) r  
Starting program: fuzzing afl-out/crashes/id:000000  
  
Program received signal SIGSEGV, Segmentation fault.  
0x0000000000000000 in ?? ()  
(gdb) bt  
#0 0x0000000000000000 in ?? ()  
#1 0x00000000004008d0 in execute (list=0x7fffffffdb10, name=0x7fffffffdb10 "pong", cmd=0x0) at fuzz.c:30  
#2 0x0000000000400a3b in main (argc=2, argv=0x7fffffffdc48) at fuzz.c:60
```



Again in `is_valid` and/or `execute`!?

```
int is_valid(functions* list, char* name) {
    int i = -1;
    while(list[++i].name) {
        if(!strncmp(list[i].name, name,
                    strlen(list[i].name))) return 1;
    }
    return 0;
}
```

```
void execute(functions* list, char*
            name, char* cmd) {
    int i = 0;
    function func;
    while(list[++i].name) {
        if(!strncmp(list[i].name, name,
                    strlen(list[i].name))) {
            func = list[i].func;
            break;
        }
    }
    func(cmd);
}
```



Again in `is_valid` and/or `execute`!?

```
int is_valid(functions* list, char* name) {  
    int i = -1;  
    while(list[++i].name) {  
        if(!strncmp(list[i].name, name,  
                    strlen(list[i].name))) return 1;  
    }  
    return 0;  
}
```

```
void execute(functions* list, char*  
            name, char* cmd) {  
    int i = 0;  
    function func;  
    while(list[++i].name) {  
        if(!strncmp(list[i].name, name,  
                    strlen(list[i].name))) {  
            func = list[i].func;  
            break;  
        }  
    }  
    func(cmd);  
}
```



Again in `is_valid` and/or `execute`!?

```
int is_valid(functions* list, char* name) {  
    int i = -1;  
    while(list[++i].name) {  
        if(!strncmp(list[i].name, name,  
                    strlen(list[i].name))) return 1;  
    }  
    return 0;  
}
```

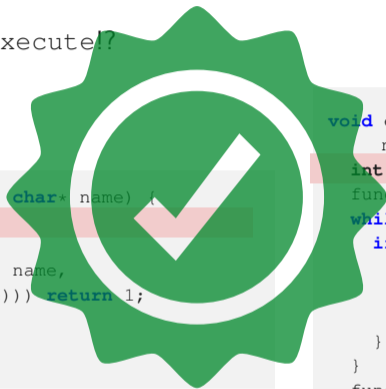
```
void execute(functions* list, char*  
            name, char* cmd) {  
    int i = -1;  
    function func;  
    while(list[++i].name) {  
        if(!strncmp(list[i].name, name,  
                    strlen(list[i].name))) {  
            func = list[i].func;  
            break;  
        }  
    }  
    func(cmd);  
}
```



Again in `is_valid` and/or `execute`!?

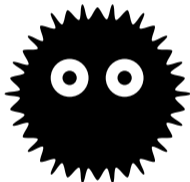
```
int is_valid(functions* list, char* name) {  
    int i = -1;  
    while(list[++i].name) {  
        if(!strncmp(list[i].name, name,  
                    strlen(list[i].name))) return 1;  
    }  
    return 0;  
}
```

```
void execute(functions* list, char*  
            name, char* cmd) {  
    int i = -1;  
    function func;  
    while(list[++i].name) {  
        if(!strncmp(list[i].name, name,  
                    strlen(list[i].name))) {  
            func = list[i].func;  
            break;  
        }  
    }  
    func(cmd);  
}
```

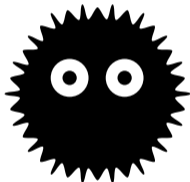




Practical Example Analysis: Fuzzing



- + Fuzzing can be **fast** and **efficient** → ≈ 4 s for 2 bugs
- + Setting up a (rather dumb) fuzzer is **easy**
- + Finds bugs that are often overlooked when manually testing



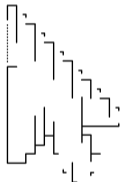
- + Fuzzing can be **fast** and **efficient** → ≈ 4 s for 2 bugs
- + Setting up a (rather dumb) fuzzer is **easy**
- + Finds bugs that are often overlooked when manually testing
- Understanding the **input** leading to a crash can be **complicated**
- Never sure if all bugs were found

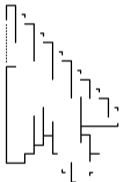


Practical Example Impact: Fuzzing

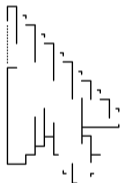


- In this case, only a NULL pointer





- In this case, only a NULL pointer
- No code execution, but attacker can crash the program



- In this case, only a NULL pointer
- No code execution, but attacker can crash the program
- Potentially dangerous if attacker can manipulate the uninitialized memory

```
typedef void (*function) (char*);  
function func;  
[...]  
func(cmd);
```



- libFuzzer fuzzes **single functions**
- Useful for **libraries**
- Easier to get code coverage
- Used in **Chromium**

THE #1 PROGRAMMER EXCUSE
FOR LEGITIMATELY SLACKING OFF:

"MY CODE'S ~~COMPILING.~~"

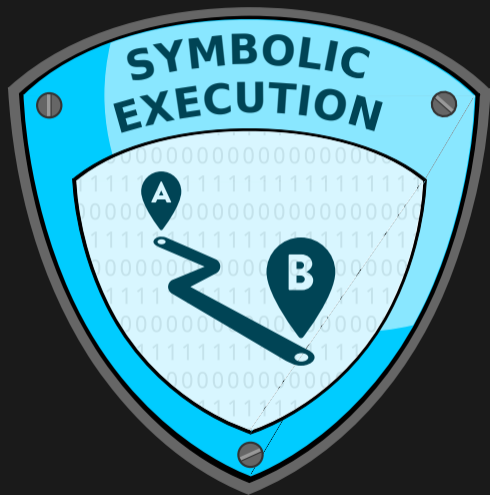
HEY! GET BACK
TO WORK!

Fuzzing!

~~COMPILING!~~

OH. CARRY ON.







- Symbolic execution finds the **required input** to reach a certain position



- Symbolic execution finds the **required input** to reach a certain position
- Programs are **interpreted**, input is modelled using **symbolic values**



- Symbolic execution finds the **required input** to reach a certain position
- Programs are **interpreted**, input is modelled using **symbolic values**
- **Variables** can be expressed using the symbolic values



- Symbolic execution finds the **required input** to reach a certain position
- Programs are **interpreted**, input is modelled using **symbolic values**
- **Variables** can be expressed using the symbolic values
- **Conditional jumps** are constraint by the symbolic values



- Symbolic execution finds the **required input** to reach a certain position
- Programs are **interpreted**, input is modelled using **symbolic values**
- **Variables** can be expressed using the symbolic values
- **Conditional jumps** are constraint by the symbolic values
- Expressions consisting of symbolic values are solved using **SAT solvers** to get concrete input

Illustration of how symbolic execution works

```
x = read ();  
y = x * 2;  
z = y + 4  
if (z == 12) {  
    bug ();  
}
```

Illustration of how symbolic execution works

```
x = read();  
y = x * 2;  
z = y + 4  
if (z == 12) {  
    bug();  
}
```

Symbolic Execution

$x = \lambda$

Illustration of how symbolic execution works

```
x = read();  
y = x * 2;  
z = y + 4  
if (z == 12) {  
    bug();  
}
```

Symbolic Execution

$y = 2 \cdot \lambda$

Illustration of how symbolic execution works

```
x = read();  
y = x * 2;  
z = y + 4  
if (z == 12) {  
    bug();  
}
```

Symbolic Execution

$z = 2 \cdot \lambda + 4$

Illustration of how symbolic execution works

```
x = read ();  
y = x * 2;  
z = y + 4  
if (z == 12) {  
    bug ();  
}
```

Symbolic Execution

SAT solver:

$$12 = 2 \cdot \lambda + 4$$

Illustration of how symbolic execution works

```
x = read ();  
y = x * 2;  
z = y + 4  
if (z == 12) {  
    bug ();  
}
```

Symbolic Execution

SAT solver:

$$12 = 2 \cdot \lambda + 4 \Rightarrow \lambda = 4$$

- Symbolic execution does not scale - possible paths grow exponentially





- Symbolic execution does not scale - possible paths grow **exponentially**
- **Unbounded loops** (*i.e.*, iterations depend on user input) have to be approximated



- Symbolic execution does not scale - possible paths grow **exponentially**
- **Unbounded loops** (*i.e.*, iterations depend on user input) have to be approximated
- **Environment** interaction is hard to model (e.g., syscalls, file system, ...)
- Possible solution: **Concolic** (concrete + symbolic) execution



- Symbolic execution does not scale - possible paths grow **exponentially**
- **Unbounded loops** (*i.e.*, iterations depend on user input) have to be approximated
- **Environment** interaction is hard to model (e.g., syscalls, file system, ...)
- Possible solution: **Concolic** (concrete + symbolic) execution
- Run symbolic execution in **parallel** with real execution, take real values if symbolic expressions get too complicated



Practical Example: Serial Number (Symbolic Execution)



```
bool checkSerial(const char *in) {
    int sum = 0;
    int digits = strlen(in);
    int parity = (digits - 1) % 2;
    for (int i = digits; i > 0; i--) {
        char current = in[i - 1];
        if (current < '0' || current > '9')
            return 0;
        int digit = current - '0';

        if (parity == i % 2)
            digit *= 2;

        sum += (digit / 10) + (digit % 10);
    }
    return 0 == sum % 10;
}
```

```
char input[256];

int main() {
    int i;
    puts("Enter verification number");
    fgets(input, 256, stdin);
    if (strlen(input) != 13)
        return 1;

    input[strlen(input) - 1] = 0;
    if (checkSerial(input)) {
        printf("Number validated!\n");
    } else {
        printf("Invalid number\n");
    }
    return 0;
}
```



```
bool checkSerial(const char *in) {
    int sum = 0;
    int digits = strlen(in);
    int parity = (digits - 1) % 2;
    for (int i = digits; i > 0; i--) {
        char current = in[i - 1];
        if (current < '0' || current > '9')
            return 0;
        int digit = current - '0';

        if (parity == i % 2)
            digit *= 2;

        sum += (digit / 10) + (digit % 10);
    }
    return 0 == sum % 10;
}
```

Input

```
char input[256];

int main() {
    int i;
    puts("Enter verification number");
    fgets(input, 256, stdin);
    if (strlen(input) != 13)
        return 1;

    input[strlen(input) - 1] = 0;
    if (checkSerial(input)) {
        printf("Number validated!\n");
    } else {
        printf("Invalid number\n");
    }
    return 0;
}
```



```
bool checkSerial(const char *in) {
    int sum = 0;
    int digits = strlen(in);
    int parity = (digits - 1) % 2;
    for (int i = digits; i > 0; i--) {
        char current = in[i - 1];
        if (current < '0' || current > '9')
            return 0;
        int digit = current - '0';

        if (parity == i % 2)
            digit *= 2;

        sum += (digit / 10) + (digit % 10);
    }
    return 0 == sum % 10;
}
```

Input `char` input[256];

```
int main() {
    int i;
    puts("Enter verification number");
    fgets(input, 256, stdin);
    if (strlen(input) != 13)
        return 1;

    input[strlen(input) - 1] = 0;
    if (checkSerial(input)) {
        printf("Number validated!\n");
    } else {
        printf("Invalid number\n");
    }
    return 0;
}
```

Go here



```
bool checkSerial(const char *in) {
    int sum = 0;
    int digits = strlen(in);
    int parity = (digits - 1) % 2;
    for (int i = digits; i > 0; i--) {
        char current = in[i - 1];
        if (current < '0' || current > '9')
            return 0;
        int digit = current - '0';

        if (parity == i % 2)
            digit *= 2;

        sum += (digit / 10) + (digit % 10);
    }
    return 0 == sum % 10;
}
```

Input `char input[256];`

```
int main() {
    int i;
    puts("Enter verification number");
    fgets(input, 256, stdin);
    if (strlen(input) != 13)
        return 1;

    input[strlen(input) - 1] = 0;
    if (checkSerial(input)) {
        printf("Number validated!\n");
    } else {
        printf("Invalid number\n");
    }
    return 0;
}
```

Go here

Avoid this



```
import angr

good = 0x8048630
avoid = (0x8048642)
length = 12

project = angr.Project('main.elf')
state = project.factory.full_init_state()
simgr = project.factory.simgr()
simgr.explore(find=good, avoid=avoid)

s = simgr.found[0]
for i in range(length):
    b = s.memory.load(0x0804a060 + i, 1)
    s.add_constraints(b >= ord('0'), b <= ord('9'))

s.se.eval_upto(s.memory.load(0x0804a060), length, length, cast_to=str)
print("Valid number: %s" % simgr.found[0].state.posix.dumps(0)[0:length])
```



Practical Example Analysis: Serial Number



```
% gcc -std=gnu99 -m32 -no-pie main.c -o main.elf
```




```
% gcc -std=gnu99 -m32 -no-pie main.c -o main.elf
% python solve.py
Valid number: 430009016964
python solve.py 25.58s user 0.88s system 100% cpu 26.204 total
```



```
% gcc -std=gnu99 -m32 -no-pie main.c -o main.elf
% python solve.py
Valid number: 430009016964
python solve.py 25.58s user 0.88s system 100% cpu 26.204 total
```

```
% ./main.elf
Enter verification number
430009016964
Number validated!
```



- Symbolic execution only took ≈ 26 s to find a valid number



- Symbolic execution only took ≈ 26 s to find a valid number
- Consider the same example with **bruteforce**:



- Symbolic execution only took ≈ 26 s to find a valid number
- Consider the same example with **bruteforce**:
 - `checkSerial` takes 700 cycles



- Symbolic execution only took ≈ 26 s to find a valid number
- Consider the same example with **bruteforce**:
 - `checkSerial` takes 700 cycles
 - 12 digits $\rightarrow 10^{12} \approx 2^{40}$ possibilities



- Symbolic execution only took ≈ 26 s to find a valid number
- Consider the same example with **bruteforce**:
 - `checkSerial` takes 700 cycles
 - 12 digits $\rightarrow 10^{12} \approx 2^{40}$ possibilities
 - 2.2 GHz CPU $\rightarrow 2.2 \cdot 10^9$ cycles / second



- Symbolic execution only took ≈ 26 s to find a valid number
- Consider the same example with **bruteforce**:
 - `checkSerial` takes 700 cycles
 - 12 digits $\rightarrow 10^{12} \approx 2^{40}$ possibilities
 - 2.2 GHz CPU $\rightarrow 2.2 \cdot 10^9$ cycles / second

$$\frac{700 \cdot 10^{12}}{2.2 \cdot 10^9} \approx 318181s \approx 88 \text{ hours}$$



Practical Example Impact: Serial Number (Symbolic Execution)



- Possible to find input to get to certain location in binary



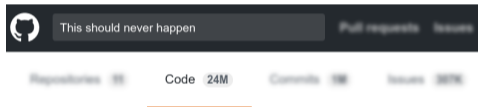


- Possible to find input to get to certain location in binary
- Find reachable location although it should not be reachable





- Possible to find input to get to certain location in binary
- Find reachable location although it should not be reachable





- Possible to find input to get to certain location in binary
- Find reachable location although it should not be reachable



Repositories 35 Code 24M Commits 196 Issues 3076

```
{  
  throw new IllegalStateException("This _really_ should  
    never happen");  
}
```



- Possible to find input to get to certain location in binary
- Find reachable location although it should not be reachable



Repositories 35 Code 24M Commits 19 Issues 3076

```
{  
  throw new IllegalStateException("This _really_ should  
    never happen");  
}
```

- Find flawed authentication (can it be bypassed?)



You are given a large binary that is hard to reverse-engineer

- You can find the challenge binary in the SSD CTF system
- It will ask you for an input and check its correctness
- If you enter the correct input you get the flag
- Use a disassembler like `radare2` and symbolic execution with `angr`
- **Hint:** Look out for endless loops, and terminate such states
- This is a semi-automated process, use `IPython` to interact with `angr`



- Symbolic execution can find **paths**, but what about **bugs**?



- Symbolic execution can find **paths**, but what about **bugs**?
- Idea: combine it with **fuzzing**



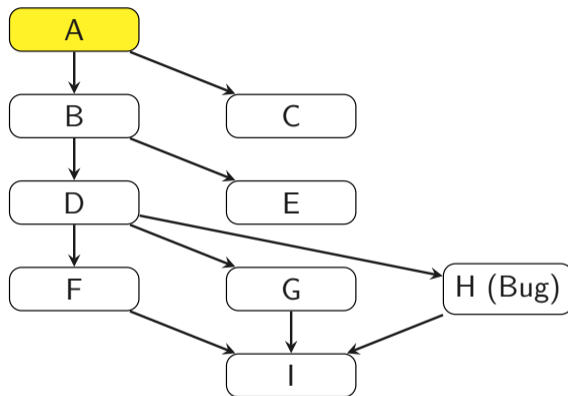
- Symbolic execution can find **paths**, but what about **bugs**?
- Idea: combine it with **fuzzing**
 - Start fuzzing
 - If fuzzer is stuck, continue with symbolic execution
 - Repeat until whole program is tested



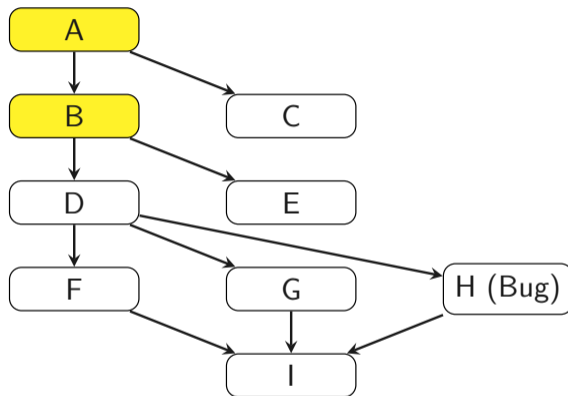
- Symbolic execution can find **paths**, but what about **bugs**?
- Idea: combine it with **fuzzing**
 - Start fuzzing
 - If fuzzer is stuck, continue with symbolic execution
 - Repeat until whole program is tested
- Combines the strengths of both approaches



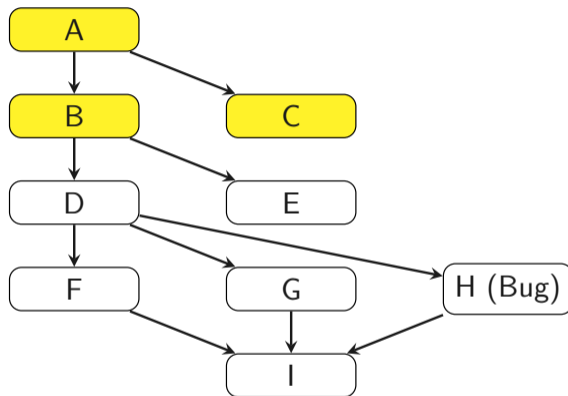
- Symbolic execution can find **paths**, but what about **bugs**?
- Idea: combine it with **fuzzing**
 - Start fuzzing
 - If fuzzer is stuck, continue with symbolic execution
 - Repeat until whole program is tested
- Combines the strengths of both approaches
- Open-source implementation **Driller**: uses AFL + angr



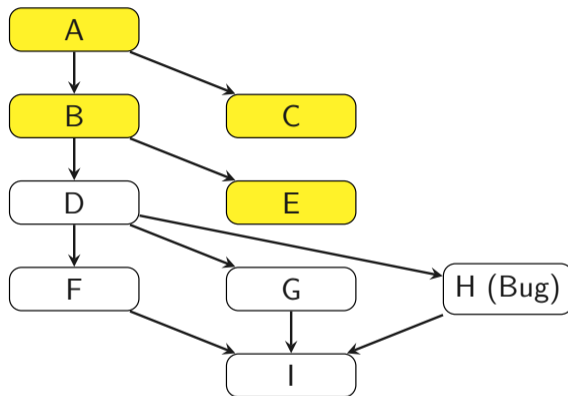
Start fuzzing



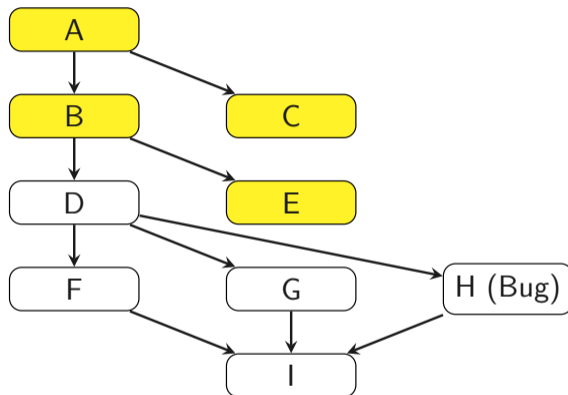
Start fuzzing



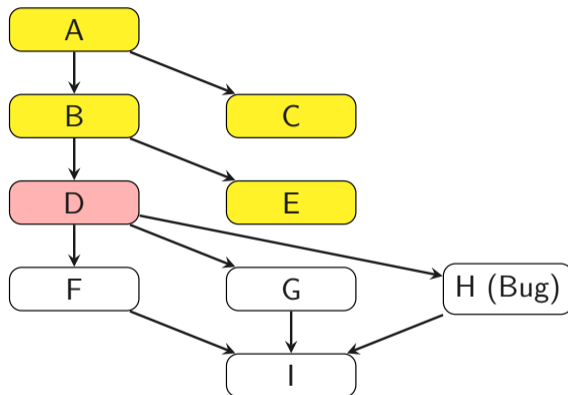
Start fuzzing



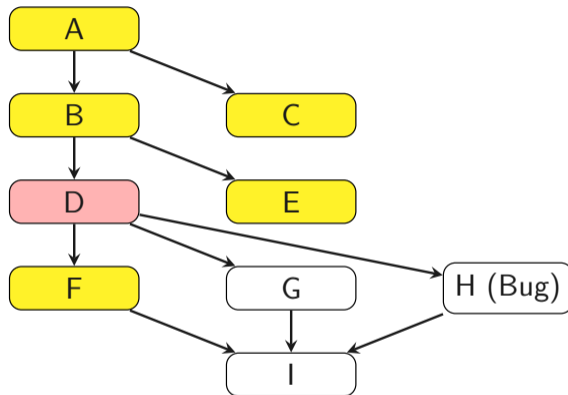
Start fuzzing



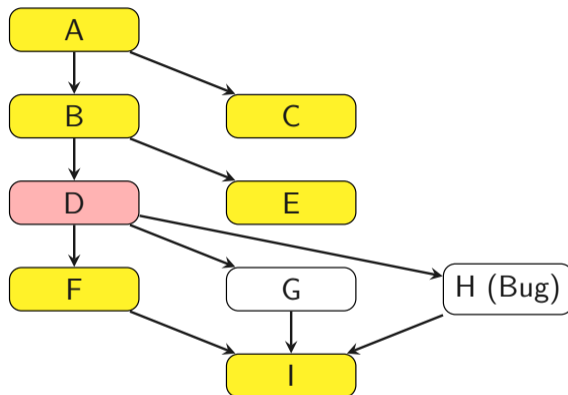
Fuzzing stuck → symbolic execution



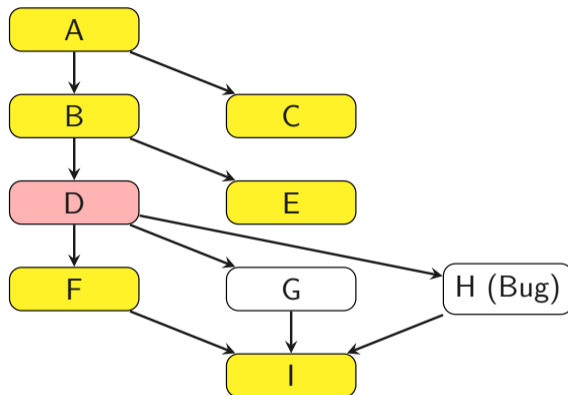
Fuzzing stuck → symbolic execution



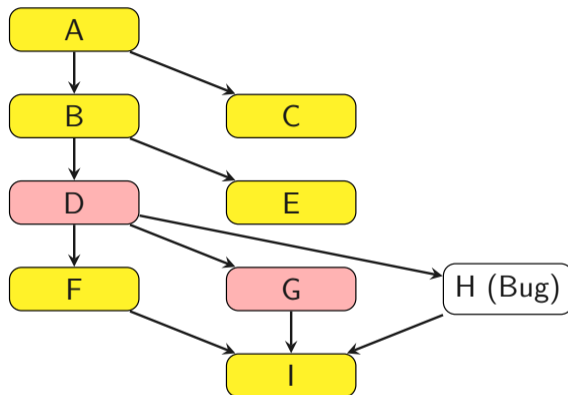
Start fuzzing (again)



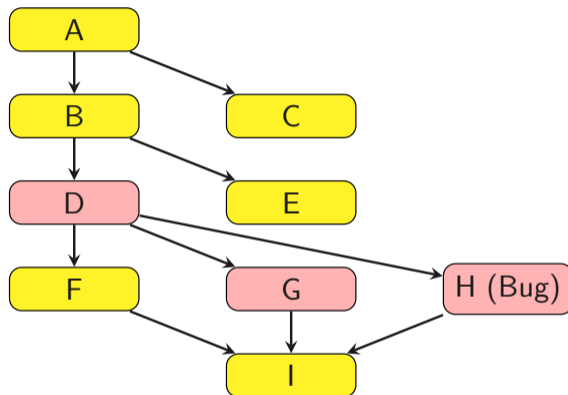
Start fuzzing (again)



Fuzzing stuck → symbolic execution



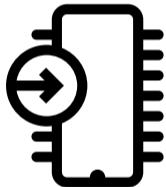
Fuzzing stuck → symbolic execution



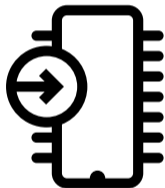
Fuzzing stuck → symbolic execution

MEMORY DEBUGGING





- **Memory Debugging** or Runtime Debugging finds memory problems
- Monitor memory accesses, allocations and deallocations



- **Memory Debugging** or Runtime Debugging finds memory problems
- Monitor memory accesses, allocations and deallocations
- Finds bug caused by wrong memory **allocation** and **deallocation**
- Can work with source code or binaries only

- **Out-of-bounds** reads/writes, e.g., buffer overflows





- **Out-of-bounds** reads/writes, e.g., buffer overflows
- Using **undefined** (*i.e.*, not initialized) values



- **Out-of-bounds** reads/writes, e.g., buffer overflows
- Using **undefined** (*i.e.*, not initialized) values
- **Incorrect frees**, e.g., double-frees, freeing memory that was not allocated



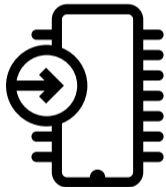
- **Out-of-bounds** reads/writes, e.g., buffer overflows
- Using **undefined** (*i.e.*, not initialized) values
- **Incorrect frees**, e.g., double-frees, freeing memory that was not allocated
- **Overlapping** memory areas for `memcpy` and similar



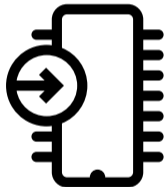
- **Out-of-bounds** reads/writes, e.g., buffer overflows
- Using **undefined** (*i.e.*, not initialized) values
- **Incorrect frees**, e.g., double-frees, freeing memory that was not allocated
- **Overlapping** memory areas for `memcpy` and similar
- **Weird values** for memory allocations (e.g., `malloc(-1)`)



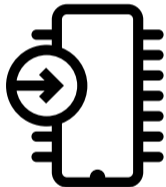
- **Out-of-bounds** reads/writes, e.g., buffer overflows
- Using **undefined** (*i.e.*, not initialized) values
- **Incorrect frees**, e.g., double-frees, freeing memory that was not allocated
- **Overlapping** memory areas for `memcpy` and similar
- **Weird values** for memory allocations (e.g., `malloc(-1)`)
- Memory **leaks**



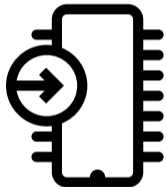
- Memory debugger has to **track** all allocations, deallocations and accesses



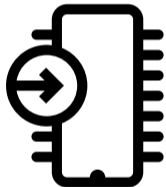
- Memory debugger has to **track** all allocations, deallocations and accesses
- Multiple approaches:
 - **Replace** the dynamic memory allocation **libraries** at compile time
 - Use **dynamic linking** (cf. LD_PRELOAD)
 - Dynamic **binary instrumentation**, *i.e.*, changing binaries at runtime



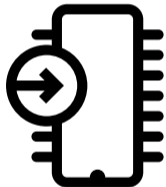
- Memory debugger has to **track** all allocations, deallocations and accesses
- Multiple approaches:
 - **Replace** the dynamic memory allocation **libraries** at compile time
 - Use **dynamic linking** (cf. LD_PRELOAD)
 - Dynamic **binary instrumentation**, *i.e.*, changing binaries at runtime
- All techniques are used in practice



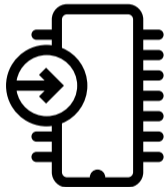
- Used in e.g., clang's address sanitizer, added directly by the compiler



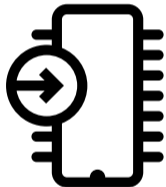
- Used in e.g., clang's address sanitizer, added directly by the compiler
- + Can find nearly all memory errors
- + Can provide detailed information, as faulty code is known



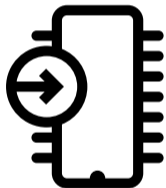
- Used in e.g., clang's address sanitizer, added directly by the compiler
- + Can find nearly all memory errors
- + Can provide detailed information, as faulty code is known
- Requires recompilation, *i.e.*, access to the source code
- Non-negligible memory and runtime overhead



- Used in e.g., Electric Fence, can be added at runtime through `LD_PRELOAD`



- Used in e.g., Electric Fence, can be added at runtime through `LD_PRELOAD`
- + No change to program required
- + No source code required



- Used in e.g., Electric Fence, can be added at runtime through `LD_PRELOAD`
- + No change to program required
- + No source code required
- Types of detectable errors are limited
- High memory overhead



- `LD_PRELOAD` can only replace library functions



- `LD_PRELOAD` can only replace library functions
- Memory reads/writes are not functions



- `LD_PRELOAD` can only replace library functions
- Memory reads/writes are not functions
- How can dynamic-linking-based techniques detect such errors?



- Use hardware/operating system support



- Use hardware/operating system support
- Allocate every buffer so it ends at a **page border**



- Use hardware/operating system support
- Allocate every buffer so it ends at a **page border**
- Do not map the next page (**guard page**)

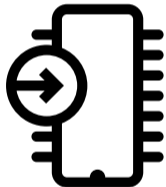




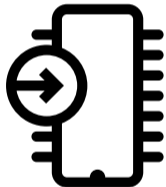
- Use hardware/operating system support
- Allocate every buffer so it ends at a **page border**
- Do not map the next page (**guard page**)



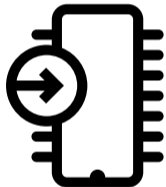
- Out-of-bounds read/write **crashes** the program (segfault)



- Used by e.g., Valgrind's memcheck



- Used by e.g., Valgrind's memcheck
- + Does not require source code
- + Can find many types of errors



- Used by e.g., Valgrind's memcheck
- + Does not require source code
- + Can find many types of errors
- Slow
- Highly architecture dependent

- Dynamic Binary Instrumentation frameworks...





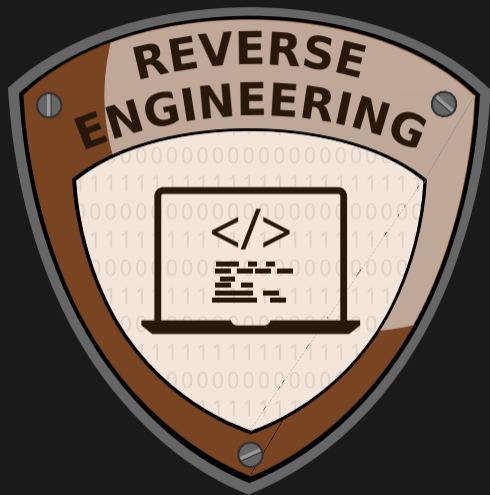
- Dynamic Binary Instrumentation frameworks...
 1. disassemble the binary
 2. add instrumentation code
 3. assemble it back

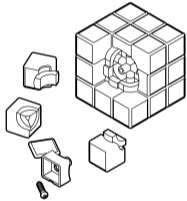


- Dynamic Binary Instrumentation frameworks...
 1. disassemble the binary
 2. add instrumentation code
 3. assemble it back
- Syscalls are wrapped to track operating system accesses to memory

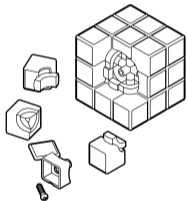


- Dynamic Binary Instrumentation frameworks...
 1. disassemble the binary
 2. add instrumentation code
 3. assemble it back
- Syscalls are wrapped to track operating system accesses to memory
- Valgrind framework supports plugins to write arbitrary instrumentation tools

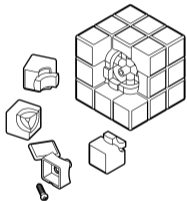




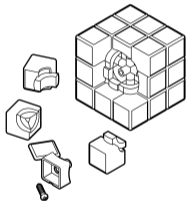
- Reverse Engineering is the process of getting back **source code** from a binary



- Reverse Engineering is the process of getting back **source code** from a binary
- Identify bugs (or hidden features) if only the binary is available



- Reverse Engineering is the process of getting back **source code** from a binary
- Identify bugs (or hidden features) if only the binary is available
- Allows to find **compiler-introduced bugs**

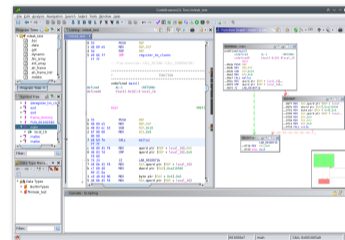
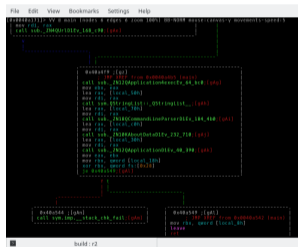
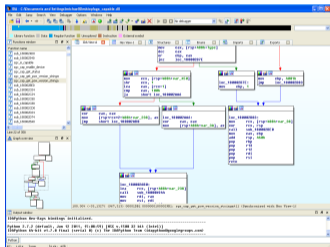


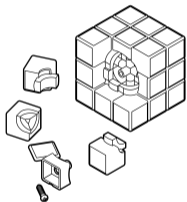
- Reverse Engineering is the process of getting back **source code** from a binary
- Identify bugs (or hidden features) if only the binary is available
- Allows to find **compiler-introduced bugs**
- Re-engineering allows to build a new binary from the reverse engineered binary

IDA Pro (≥ 1200 €)

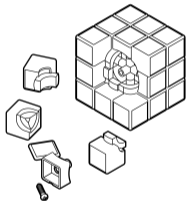
radare2 (open source)

Ghidra (open source)

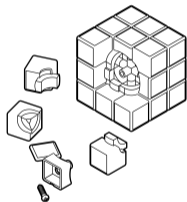




- Disassembler allows to
 - **disassemble** code (get assembly code)
 - **analyze** binaries (dependencies, strings, control flow)
 - **debug** programs (see actual register values, step through code)



- Disassembler allows to
 - **disassemble** code (get assembly code)
 - **analyze** binaries (dependencies, strings, control flow)
 - **debug** programs (see actual register values, step through code)
- **Decompilers** further convert code to high-level language (C or pseudo code)



- Disassembler allows to
 - **disassemble** code (get assembly code)
 - **analyze** binaries (dependencies, strings, control flow)
 - **debug** programs (see actual register values, step through code)
- **Decompilers** further convert code to high-level language (C or pseudo code)
- Good decompilers cost a lot of money ☹️ (IDA Pro - HexRays)



Practical Example: Disassembly vs. Decompilation



```
#include <stdio.h>
#include <string.h>

int main() {
    char buffer[64];
    printf("Enter password:\n");
    fgets(buffer, 64, stdin);
    if(!strncmp(buffer, "secret1234", 10)) {
        printf("Correct!\n");
    } else {
        printf("Wrong\n");
    }
    return 0;
}
```



```
#include <stdio.h>
#include <string.h>

int main() {
    char buffer[64];
    printf("Enter password:\n");
    fgets(buffer, 64, stdin);
    if(!strncmp(buffer, "secret1234", 10)) {
        printf("Correct!\n");
    } else {
        printf("Wrong\n");
    }
    return 0;
}
```

```
% gcc re.c -o re
```



Practical Example Analysis: Disassembly vs. Decompilation



```
% r2 re
[0x00400500]> aaaa
[x] Analyze all flags starting with sym. and entry0 (aa)
[x] Analyze len bytes of instructions for references (aar)
[x] Analyze function calls (aac)
[x] Emulate code to find computed references (aae)
[x] Analyze consecutive function (aat)
[x] Constructing a function name for fcn.* and
    sym.func.* functions (aan)
[x] Type matching analysis for all functions (afta)
[0x00400500]> VV @ main
```



[0x04005E6]> VV @ main (nodes 4 edges 4 zoom 100%) BB-NCM mouse:canvas-y movements-speed:5

```
[0x4005E6] : [gc]
j=main
(func) main 97
main: 0;
; var int local_4Ch @ rbp-0x40
; DATA XREF from 0x04005Dd (str.zy0)
push rbp
mov rbp, rsp
; 0
sub rbp, 0x40
; const char * a
; 0x4006e4
; "Enter password:"
mov edi, str.Enter.password:
call sym.imp.puts:[gc]
; r14, r14name
; 0x001050-8)=0
mov r14, qword [rbp.stdia]
lea rax, [local_4Ch]
] rax r14
sar byte [r14 + 0x40], 0x48
mov edi, eax
call sym.imp.puts:[gc]
lea rax, [local_4Ch]
; s1=2 n
mov edi, 0x2
; const char * s2
; 0x4006e4
; "secret1234"
mov esi, str.secret1234
; const char * s1
mov edi, rax
call sym.imp.strncmp:[gc]
test eax, eax
jnz 0x400646:[gd]
```





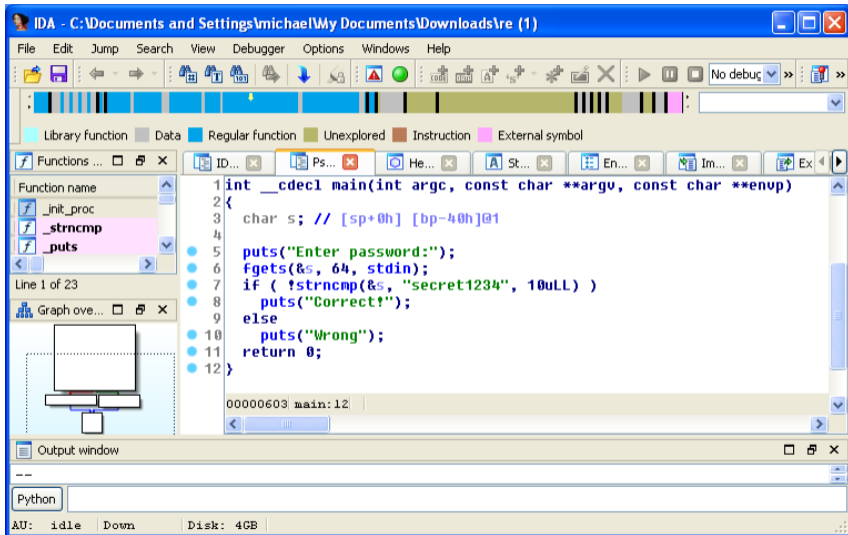
```
call sym.imp.fgets; [gb]
lea rax, [local_40h]
; size_t n
mov ecx, 0xa
; const char * s2
; 0x4006f4
; "secret1234"
mov esi, str.secret1234
; const char * s1
mov rdi, rax
call sym.imp.strncmp; [gc]
test eax, eax
jne 0x400646; [gd]
```

f t

```
0x40063a ; [gg]
; const char * s
; 0x4006ff
; "Correct!"
mov edi, str.Correct_
call sym.imp.puts; [ga]
jmp 0x400650; [gf]
```

```
0x400646 ; [gd]
; const char * s
; 0x400708
; "Wrong"
mov edi, str.Wrong
call sym.imp.puts; [ga]
```







The screenshot shows the CodeBrowser application window titled "CodeBrowser: Test:/re". The main editor displays the decompiled C code for the "main" function. The code is as follows:

```
1  undefined8 main(void)
2
3
4  {
5      int iVar1;
6      char local_48 [64];
7
8      puts("Enter password:");
9      fgets(local_48,0x40,stdin);
10     iVar1 = strcmp(local_48,"secret1234",10);
11     if (iVar1 == 0) {
12         puts("Correct!");
13     }
14     else {
15         puts("Wrong");
16     }
17     return 0;
18 }
19
```

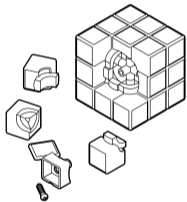
The left sidebar contains several panels: "Program Trees" showing a tree view with "re" and ".bss"; "Symbol Tree" showing symbols "f frame_d" and "f FUN_00"; and "Data Ty...". The top menu bar includes "File", "Edit", "Analysis", "Navigation", "Search", "Select", "Tools", "Window", and "Help".



Practical Example Impact: Disassembly vs. Decompilation

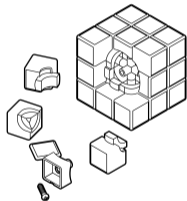


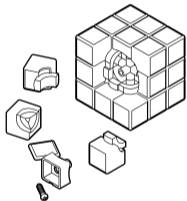
- Disassembler only returns often hard-to-understand assembly code



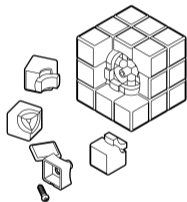


- Disassembler only returns often hard-to-understand assembly code
- Decompilation output is often a lot **easier to read**

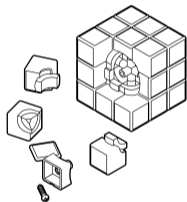




- Disassembler only returns often hard-to-understand assembly code
- Decompilation output is often a lot **easier to read**
- However, decompilation is a lot of **magic** - does not always work



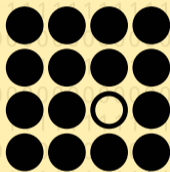
- Disassembler only returns often hard-to-understand assembly code
- Decompilation output is often a lot **easier to read**
- However, decompilation is a lot of **magic** - does not always work
- Highly dependent on
 - architecture
 - used compiler
 - optimization level
 - obfuscation



- Disassembler only returns often hard-to-understand assembly code
- Decompilation output is often a lot **easier to read**
- However, decompilation is a lot of **magic** - does not always work
- Highly dependent on
 - architecture
 - used compiler
 - optimization level
 - obfuscation
- If it works, it gives a quick **overview** for further investigations



**BINARY
DIFFING**





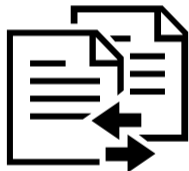
- Security patches for closed-source products often have no (real) description of the bug



- Security patches for closed-source products often have no (real) **description of the bug**
- The patch is usually available for download



- Security patches for closed-source products often have no (real) **description of the bug**
- The patch is usually available for download
- Binary diffing is like **normal diffing**, except for binaries



- Security patches for closed-source products often have no (real) **description of the bug**
- The patch is usually available for download
- Binary diffing is like **normal diffing**, except for binaries
- Reveals differences in two binaries, *i.e.*, the bug fix



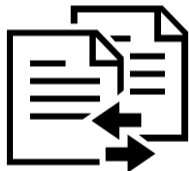
- Security patches for closed-source products often have no (real) **description of the bug**
- The patch is usually available for download
- Binary diffing is like **normal diffing**, except for binaries
- Reveals differences in two binaries, *i.e.*, the bug fix
- Can also be used to find **vulnerable functions** by comparing binary with known vulnerable functions

- Diffing tools use different methods to find matching and unmatching blocks





- Diffing tools use different methods to find matching and unmatching blocks
 - Same function **name**
 - Same **assembly**, same decompiled code
 - Equal number of **calls** to and from function
 - Same referenced **strings**
 - ...



- Diffing tools use different methods to find matching and unmatching blocks
 - Same function **name**
 - Same **assembly**, same decompiled code
 - Equal number of **calls** to and from function
 - Same referenced **strings**
 - ...
- Most diff tools rely on the **control-flow graph** from a disassembler



- Diffing tools use different methods to find matching and unmatching blocks
 - Same function **name**
 - Same **assembly**, same decompiled code
 - Equal number of **calls** to and from function
 - Same referenced **strings**
 - ...
- Most diff tools rely on the **control-flow graph** from a disassembler
- The **basic blocks** are then matched using some heuristic



Practical Example: 1-day (Binary Diffing)



```
[Da@040120]~$ W & main (nodes 4 edges 4 score 101) 0x-0x200 mouse;0x4000-y;movements;speed;5

[0x01720] : [gb]
;-- main
[0x01 main] 239
main ()
; var int local_90h @ stp-0x98
; var char local_30h @ stp-0x50
; var int local_80h @ stp-0x10
; var int local_90h @ stp-0x80
; var int local_02h @ stp-0x60
; var int local_10h @ stp-0x40
; var int local_5h @ stp-0x0
; var int local_4h @ stp-0x0
; call 380 from 0x010064 (entry0)
push ebp
mov esp, esp
sub esp, 0x10
mov dword [local_0h], 0
mov dword [local_8h], 0x1
mov dword [local_10h], 0x1
mov
; [0x01] = 1
call dword [local_8h], 2
; 0x400740 [gb]

[0x01743] : [gb]
; var eax, eax
; char * const str
; var esi, eax
; int i
; var eax, eax
; mov ecx, dword [local_10h]
; const char * str
; [0x01] = 1
; i
; mov esi, dword [ecx + 0]
; call esp, getrand, [gb]
; jmp esi, [local_00h]
; jmp esi, [local_00h]
; mov ecx, [local_00h], eax
; mov esi, dword [local_80h]
; mov ecx, dword [local_80h]
; mov esi, dword [local_80h]
; mov ecx, dword [local_30h], esi
; mov esi, eax
; mov dword [local_80h], ecx
; mov dword [stp - 0x04], ecx
; call esp, getrand, [gb]
; const char *
; dword esi, 0x410033
; char * s
; mov esi, dword [local_30h]
;
; mov ecx, dword [stp - 0x04]
; mov ecx, dword [local_30h]
; mov si, eax
; mov al, 0
; call esp, getrand, [gb]
; const char * format
; dword esi, 0x410033
; jmp esi, [local_00h]
; mov dword [stp - 0x04], eax
; mov al, 0
; call esp, getrand, [gb]
; mov dword [stp - 0x04], eax
; jmp 0x400740 [gb]

[0x400740] : [gb]
; const char * format
; 0x400740
; "temp: %s\n\n\n"
; mov esi, dword [local_10h]
; mov ecx, dword [eax]
; mov al, 0
; call esp, getrand, [gb]
; mov dword [stp - 0x04], eax

[0x410034] : [gb]
; jmp 0x007 from 0x400740, 0x410034
```

- Given a binary with an **unknown bug**
- The vendor released a **patch**, fixing the vulnerability



```
[0x0400120]> W @ main (nodes 4 edges 4 score 101) 0x-0000 mouse:0x0400120-movementa-speed:5

[0x041720] : [gb]
;-- main
[0x0] main 239
main ()
; var int local_90h @ stp-0x98
; var char local_30h @ stp-0x50
; var int local_80h @ stp-0x10
; var int local_90h @ stp-0x80
; var int local_02h @ stp-0x00
; var int local_10h @ stp-0x40
; var int local_5h @ stp-0x40
; var int local_4h @ stp-0x40
; call 380 from 0x0410064 (entry0)
push ebp
mov esp, esp
sub esp, 0x10
mov dword [local_0h], 0
mov dword [local_8h], edi
mov dword [local_10h], esi
xor
; [0x041]=1
cmp dword [local_0h], 2
jne 0x007340 [gg]

[0x041743] : [gb]
; var eax, eax
; char *const
; mov esi, eax
; int base
; var esi, eax
; mov ecx, dword [local_10h]
; const char * str
; [0x041]=1
;
; mov esi, dword [ecx + 0]
; call sym_imp_getintf [gg]
; jmp esi, [local_00h]
; jmp esi, [local_00h]
; mov ecx, [local_00h], eax
; mov esi, dword [local_80h]
; mov ecx, dword [local_80h]
; mov esi, dword [local_80h]
; mov ecx, dword [local_30h], esi
; mov esi, eax
; mov dword [local_80h], ecx
; mov dword [stp - 0x04], ecx
; call sym_imp_getintf [gg]
; const char
; dword esi, 0x410038
; char *s
; mov esi, dword [local_30h]
;
; mov ecx, dword [stp - 0x04]
; mov ecx, dword [local_30h]
; mov si, eax
; mov al, 0
; call sym_imp_getintf [gg]
; const char * format
; dword esi, 0x410038
; jmp esi, [local_00h]
; mov dword [stp - 0x04], eax
; mov al, 0
; call sym_imp_getintf [gg]
; mov dword [stp - 0x04], eax
; jmp 0x007340 [gg]

[0x007340] : [gg]
; const char * format
; 0x0038
; const char *format, ..s
; mov esi, dword [local_10h]
; mov ecx, dword [eax]
; mov al, 0
; call sym_imp_getintf [gg]
; mov dword [stp - 0x04], eax
; jmp 0x007340 [gg]
```

- Given a binary with an **unknown bug**
- The vendor released a **patch**, fixing the vulnerability
- Fixed bug has **no documentation...**



```
% ./vuln  
Usage: ./vuln <number>
```



```
% ./vuln
Usage: ./vuln <number>
```

```
% ./vuln 123
Dec: 123
Hex: 0x7b
Bin: 0b1111011
```




```
% radiff2 -AAAAC vuln patched
```



```
% radiff2 -AAAAC vuln patched
```

```
sym._init 26 0x4004a8 | UNMATCH (0.923077) | 0x4004e0 26 sym._init
sym.imp.strcpy 32 0x4004e0 | UNMATCH (0.906250) | 0x400510 32 sym.imp.strcpy
sym.imp.printf 48 0x4004f0 | UNMATCH (0.854167) | 0x400530 48 sym.imp.printf
sym.imp.__libc_start_main 48 0x400500 | UNMATCH (0.854167) | 0x400550 48 sym.imp.__libc_start_main
sym.imp.strtol 48 0x400510 | UNMATCH (0.854167) | 0x400560 48 sym.imp.strtol
sym.imp.strcat 48 0x400520 | UNMATCH (0.854167) | 0x400520 48 sym.imp.strlen
sym.imp.sprintf 48 0x400530 | UNMATCH (0.854167) | 0x400570 48 sym.imp.sprintf
fcn.00400540 57 0x400540 | MATCH (0.192982) | 0x400580 57 fcn.00400580
sym.deregister_tm_clones 35 0x400580 | UNMATCH (0.914286) | 0x4005c0 35 sym.deregister_tm_clones
sym.to_bin 236 0x400630 | MATCH (0.723776) | 0x400670 286 sym.to_bin
sym.imp.strncat 48 0x400540 | NEW (0.000000)
```



```
% radiff2 -AAAAC vuln patched
```

| | | | | | | | | | |
|---------------------------|-----|----------|--|---------|------------|--|----------|-----|---------------------------|
| sym._init | 26 | 0x4004a8 | | UNMATCH | (0.923077) | | 0x4004e0 | 26 | sym._init |
| sym.imp.strcpy | 32 | 0x4004e0 | | UNMATCH | (0.906250) | | 0x400510 | 32 | sym.imp.strcpy |
| sym.imp.printf | 48 | 0x4004f0 | | UNMATCH | (0.854167) | | 0x400530 | 48 | sym.imp.printf |
| sym.imp.__libc_start_main | 48 | 0x400500 | | UNMATCH | (0.854167) | | 0x400550 | 48 | sym.imp.__libc_start_main |
| sym.imp.strtol | 48 | 0x400510 | | UNMATCH | (0.854167) | | 0x400560 | 48 | sym.imp.strtol |
| sym.imp.strcat | 48 | 0x400520 | | UNMATCH | (0.854167) | | 0x400520 | 48 | sym.imp.strlen |
| sym.imp.sprintf | 48 | 0x400530 | | UNMATCH | (0.854167) | | 0x400570 | 48 | sym.imp.sprintf |
| fcn.00400540 | 57 | 0x400540 | | MATCH | (0.192982) | | 0x400580 | 57 | fcn.00400580 |
| sym.deregister_tm_clones | 35 | 0x400580 | | UNMATCH | (0.914286) | | 0x4005c0 | 35 | sym.deregister_tm_clones |
| sym.to_bin | 236 | 0x400630 | | MATCH | (0.723776) | | 0x400670 | 286 | sym.to_bin |
| sym.imp.strncat | 48 | 0x400540 | | NEW | (0.000000) | | | | |



Practical Example Analysis: 1-day (Binary Diffing)



```
% radiff2 -g sym.to_bin vuln patched | xdot -
```

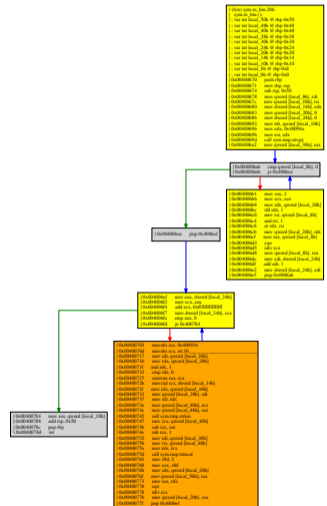
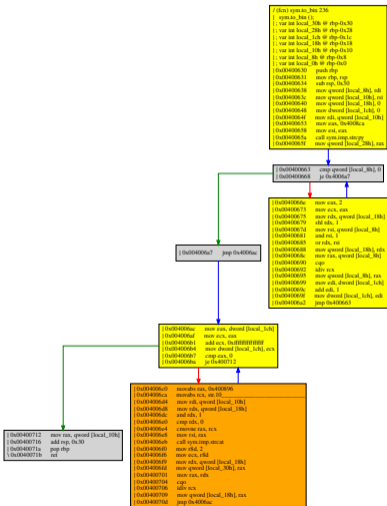
```
% radiff2 -g sym.to_bin patched vuln | xdot -
```

1-day (Binary Diffing)



```
% radiff2 -g sym.to_bin vuln patched | xdot -
```

```
% radiff2 -g sym.to_bin patched vuln | xdot -
```





```
| 0x004006c0 movabs rax, 0x400896
| 0x004006ca movabs rcx, str.10
| 0x004006d4 mov rdi, qword [local_10h]
| 0x004006d8 mov rdx, qword [local_18h]
| 0x004006dc and rdx, 1
| 0x004006e0 cmp rdx, 0
| 0x004006e4 cmovne rax, rcx
| 0x004006e8 mov rsi, rax
| 0x004006eb call sym.imp.strcat
| 0x004006f0 mov r8d, 2
| 0x004006f6 mov ecx, r8d
| 0x004006f9 mov rdx, qword [local_18h]
| 0x004006fd mov qword [local_30h], rax
| 0x00400701 mov rax, rdx
| 0x00400704 cqo
| 0x00400706 idiv rcx
| 0x00400709 mov qword [local_18h], rax
| 0x0040070d jmp 0x4006ac
```

Original (vuln)

```
| 0x00400703 movabs rax, 0x400916
| 0x0040070d movabs rcx, str.10
| 0x00400717 mov rdi, qword [local_10h]
| 0x0040071b mov rdx, qword [local_20h]
| 0x0040071f and rdx, 1
| 0x00400723 cmp rdx, 0
| 0x00400727 cmovne rax, rcx
| 0x0040072b movsxd rcx, dword [local_14h]
| 0x0040072f mov rdx, qword [local_10h]
| 0x00400733 mov qword [local_38h], rdi
| 0x00400737 mov rdi, rdx
| 0x0040073a mov qword [local_40h], rcx
| 0x0040073e mov qword [local_48h], rax
| 0x00400742 call sym.imp.strlen
| 0x00400747 mov rcx, qword [local_40h]
| 0x0040074b sub rcx, rax
| 0x0040074e sub rcx, 1
| 0x00400752 mov rdi, qword [local_38h]
| 0x00400756 mov rsi, qword [local_48h]
| 0x0040075a mov rdx, rcx
| 0x0040075d call sym.imp.strncat
| 0x00400762 mov r8d, 2
| 0x00400768 mov ecx, r8d
| 0x0040076b mov rdx, qword [local_20h]
| 0x0040076f mov qword [local_50h], rax
| 0x00400773 mov rax, rdx
| 0x00400776 cqo
| 0x00400778 idiv rcx
| 0x0040077b mov qword [local_20h], rax
| 0x0040077f jmp 0x4006ef
```

Patched (patched)



```
| 0x004006c0 movabs rax, 0x400896
| 0x004006ca movabs rcx, str.10
| 0x004006d4 mov rdi, qword [local_10h]
| 0x004006d8 mov rdx, qword [local_18h]
| 0x004006dc and rdx, 1
| 0x004006e0 cmp rdx, 0
| 0x004006e4 cmovne rax, rcx
| 0x004006e8 mov rsi, rax
| 0x004006eb call sym.imp.strcat
| 0x004006f0 mov r8d, 2
| 0x004006f6 mov ecx, r8d
| 0x004006f9 mov rdx, qword [local_18h]
| 0x004006fd mov qword [local_30h], rax
| 0x00400701 mov rax, rdx
| 0x00400704 cqo
| 0x00400706 idiv rcx
| 0x00400709 mov qword [local_18h], rax
| 0x0040070d jmp 0x4006ac
```

Bad: strcat

Original (vuln)

```
| 0x00400703 movabs rax, 0x400916
| 0x0040070d movabs rcx, str.10
| 0x00400717 mov rdi, qword [local_10h]
| 0x0040071b mov rdx, qword [local_20h]
| 0x0040071f and rdx, 1
| 0x00400723 cmp rdx, 0
| 0x00400727 cmovne rax, rcx
| 0x0040072b movsxd rcx, dword [local_14h]
| 0x0040072f mov rdx, qword [local_10h]
| 0x00400733 mov qword [local_38h], rdi
| 0x00400737 mov rdi, rdx
| 0x0040073a mov qword [local_40h], rcx
| 0x0040073e mov qword [local_48h], rax
| 0x00400742 call sym.imp.strlen
| 0x00400747 mov rcx, qword [local_40h]
| 0x0040074b sub rcx, rax
| 0x0040074e sub rcx, 1
| 0x00400752 mov rdi, qword [local_38h]
| 0x00400756 mov rsi, qword [local_48h]
| 0x0040075a mov rdx, rcx
| 0x0040075d call sym.imp.strncat
| 0x00400762 mov r8d, 2
| 0x00400768 mov ecx, r8d
| 0x0040076b mov rdx, qword [local_20h]
| 0x0040076f mov qword [local_50h], rax
| 0x00400773 mov rax, rdx
| 0x00400776 cqo
| 0x00400778 idiv rcx
| 0x0040077b mov qword [local_20h], rax
| 0x0040077f jmp 0x4006ef
```

Patched (patched)



```
| 0x004006c0 movabs rax, 0x400896
| 0x004006ca movabs rcx, str.10
| 0x004006d4 mov rdi, qword [local_10h]
| 0x004006d8 mov rdx, qword [local_18h]
| 0x004006dc and rdx, 1
| 0x004006e0 cmp rdx, 0
| 0x004006e4 cmovne rax, rcx
| 0x004006e8 mov rsi, rax
| 0x004006eb call sym.imp.strcat
| 0x004006f0 mov r8d, 2
| 0x004006f6 mov ecx, r8d
| 0x004006f9 mov rdx, qword [local_18h]
| 0x004006fd mov qword [local_30h], rax
| 0x00400701 mov rax, rdx
| 0x00400704 cqo
| 0x00400706 idiv rcx
| 0x00400709 mov qword [local_18h], rax
| 0x0040070d jmp 0x4006ac
```

Bad: strcat

Original (vuln)

```
| 0x00400703 movabs rax, 0x400916
| 0x0040070d movabs rcx, str.10
| 0x00400717 mov rdi, qword [local_10h]
| 0x0040071b mov rdx, qword [local_20h]
| 0x0040071f and rdx, 1
| 0x00400723 cmp rdx, 0
| 0x00400727 cmovne rax, rcx
| 0x0040072b movsxd rcx, dword [local_14h]
| 0x0040072f mov rdx, qword [local_10h]
| 0x00400733 mov qword [local_38h], rdi
| 0x00400737 mov rdi, rdx
| 0x0040073a mov qword [local_40h], rcx
| 0x0040073e mov qword [local_48h], rax
| 0x00400742 call sym.imp.strlen
| 0x00400747 mov rcx, qword [local_40h]
| 0x0040074b sub rcx, rax
| 0x0040074e sub rcx, 1
| 0x00400752 mov rdi, qword [local_38h]
| 0x00400756 mov rsi, qword [local_48h]
| 0x0040075a mov rdx, rcx
| 0x0040075d call sym.imp.strncat
| 0x00400762 mov r8d, 2
| 0x00400768 mov ecx, r8d
| 0x0040076b mov rdx, qword [local_20h]
| 0x0040076f mov qword [local_50h], rax
| 0x00400773 mov rax, rdx
| 0x00400776 cqo
| 0x00400778 idiv rcx
| 0x0040077b mov qword [local_20h], rax
| 0x0040077f jmp 0x4006ef
```

Good: strncat

Patched (patched)



```
% ./vuln 9999999999
Dec: 9999999999
Hex: 0x2540be3ff
Bin: 0b10010101000000101111100011111111Dec: 9999999999
Hex: 0x2540be3ff
Bin: 0b100

[1]      27986 segmentation fault (core dumped)  ./vuln 9999999999
```



```
% ./vuln 9999999999
Dec: 9999999999
Hex: 0x2540be3ff
Bin: 0b10010101000000101111100011111111Dec: 9999999999
Hex: 0x2540be3ff
Bin: 0b100

[1]      27986 segmentation fault (core dumped)  ./vuln 9999999999
```

```
% ./patched 9999999999
Dec: 9999999999
Hex: 0x2540be3ff
Bin: 0b100101010000001011111000111111
```



Practical Example Impact: 1-day (Binary Diffing)



- Binary diffing is a way to reverse engineer patches





- Binary diffing is a way to reverse engineer patches
- If there are not many changes, vulnerability can be quickly spotted



- Binary diffing is a way to reverse engineer **patches**
- If there are not many changes, vulnerability can be quickly spotted
- Knowledge of the vulnerability allows attackers to craft **exploits**



- Binary diffing is a way to reverse engineer **patches**
- If there are not many changes, vulnerability can be quickly spotted
- Knowledge of the vulnerability allows attackers to craft **exploits**
- As long as patches are not applied, such **1-days** are effective



- Binary diffing is a way to reverse engineer **patches**
- If there are not many changes, vulnerability can be quickly spotted
- Knowledge of the vulnerability allows attackers to craft **exploits**
- As long as patches are not applied, such **1-days** are effective
- Also a starting point for same/similar bugs in the program



Real-world Example: Apple's Password Hint Bug



- macOS Sierra had a bug in the **file-system encryption**



- macOS Sierra had a bug in the **file-system encryption**
- User can set a **password hint** which can be displayed



- macOS Sierra had a bug in the **file-system encryption**
- User can set a **password hint** which can be displayed
- However, the password **hint** was **never shown**, but...



- macOS Sierra had a bug in the **file-system encryption**
- User can set a **password hint** which can be displayed
- However, the password **hint** was **never shown**, but...
- ...the **password was shown** instead





Security experts used **binary diffing** on the patch...

```
1 if ( v50 )
2     objc_msgSend(v19, "setObject:forKey:", v50, CFSTR("kSKAPFSDiskPasswordOption"));
3     if ( a9 )
4         objc_msgSend(v19, "setObject:forKey:", v50, CFSTR("kSKAPFSDiskPasswordHintOption"));
```

U:--- OldStorageKit.txt All (5,0) [i] (ObjC/1 FlyC- Projectile[-] SP/s)



Security experts used **binary diffing** on the patch...

```
1 if ( v50 )
2   objc_msgSend(v19, "setObject:forKey:", v50, CFSTR("kSKAPFSDiskPasswordOption"));
3   if ( a9 )
4     objc_msgSend(v19, "setObject:forKey:", v50, CFSTR("kSKAPFSDiskPasswordHintOption"));
```

U:--- OldStorageKit.txt All (5,0) [i] (ObjC/l FlyC- Projectile[-] SP/s)

...to discover that it was a **copy&paste fail**

```
1 if ( v51 )
2   objc_msgSend(v19, "setObject:forKey:", v51, CFSTR("kSKAPFSDiskPasswordOption"));
3   if ( v49 )
4     objc_msgSend(v19, "setObject:forKey:", v49, CFSTR("kSKAPFSDiskPasswordHintOption"));
```

U:--- NewStorageKit.txt All (5,0) [i] (ObjC/l FlyC- Projectile[-] SP/s)



- Finding bugs is like catching fish - you don't know how long it takes





- Finding bugs is like catching fish - you don't know how long it takes
- Hard part is to figure out where the fish (bug) is, and when it will bite (occur)



- Finding bugs is like catching fish - you don't know how long it takes
- Hard part is to figure out where the fish (bug) is, and when it will bite (occur)
- There are many **tools supporting** developers in finding bugs



- Finding bugs is like catching fish - you don't know how long it takes
- Hard part is to figure out where the fish (bug) is, and when it will bite (occur)
- There are many **tools supporting** developers in finding bugs
- Some of them are completely **automated** - use them!



- Finding bugs is like catching fish - you don't know how long it takes
- Hard part is to figure out where the fish (bug) is, and when it will bite (occur)
- There are many **tools supporting** developers in finding bugs
- Some of them are completely **automated** - use them!
- **Incorporate** bug finding **tools** into your **development process**



- Finding bugs is like catching fish - you don't know how long it takes
- Hard part is to figure out where the fish (bug) is, and when it will bite (occur)
- There are many **tools supporting** developers in finding bugs
- Some of them are completely **automated** - use them!
- **Incorporate** bug finding **tools** into your **development process**
- It **does not cost a lot** to compile with sanitizers, run static code analysis, fuzz your software, ...



- Finding bugs is like catching fish - you don't know how long it takes
- Hard part is to figure out where the fish (bug) is, and when it will bite (occur)
- There are many **tools supporting** developers in finding bugs
- Some of them are completely **automated** - use them!
- **Incorporate** bug finding **tools** into your **development process**
- It **does not cost a lot** to compile with sanitizers, run static code analysis, fuzz your software, ...
- ...but it **eliminates many bugs** (for free)

Questions?


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

```
int main(int argc, char* argv[]) {  
    printf(argv[0]);  
    return 0;  
}
```

**There are no bugs,
just happy little accidents**



-  Nicholas Nethercote and Julian Seward.
Valgrind: a framework for heavyweight dynamic binary instrumentation.
In ACM Sigplan notices, 2007.
-  Jeongwook Oh.
Fight against 1-day exploits: Diffing binaries vs anti-diffing binaries.
Black Hat, 2009.
-  Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna.
SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis.
In IEEE Symposium on Security and Privacy, 2016.

-  Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna.
Driller: Augmenting fuzzing through selective symbolic execution.
In NDSS, 2016.