

Secure Software Development

Finding Bugs I

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

18.11.2022

Winter 2022/23, www.iaik.tugraz.at

1. Introduction
2. Human Expert
3. Static Analysis
4. Sanitizers
5. Debugging

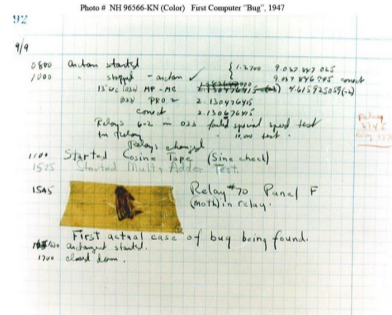
Introduction

- “Bug” used as a term for malfunctions since 1870s

- “Bug” used as a term for malfunctions since 1870s
- Asimov 1944: bugs are issues with robots (I, Robot)

- “Bug” used as a term for malfunctions since 1870s
- Asimov 1944: bugs are issues with robots (I, Robot)
- First computer bug report

- “Bug” used as a term for malfunctions since 1870s
- Asimov 1944: bugs are issues with robots (I, Robot)
- First computer bug report 09.09.1947




- “Bug” used as a term for malfunctions since 1870s
- Asimov 1944: bugs are issues with robots (I, Robot)
- First computer bug report 09.09.1947
 - Grace Hopper’s team

Photo # NH 96566-KN (Color) First Computer “Bug”, 1947

9/9

0800 Antman started
1000 - stopped - antman ✓ { 1.2700 9.057 867 025
13'00 1000 HP - MC 1.3047695 9.057 867 025
0200 PRO = 2.13047695 9.057 867 025
0200 CON = 2.13047695 9.057 867 025
Relays 6-2 in 022 fault speed speed test
in relay = new test.

1100 Relays changed
Started Cosine Tape (Sine check)
1525 Started Multi Adder Test.

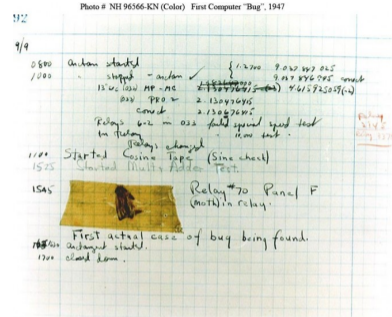
1545  Relay #70 Panel F
(moth) in relay.

1700 First actual case of bug being found.
1700 antman started.
1700 clock down.

Relay #70
2145
0200 9337



- “Bug” used as a term for malfunctions since 1870s
- Asimov 1944: bugs are issues with robots (I, Robot)
- First computer bug report 09.09.1947
 - Grace Hopper’s team
- “Debugging” aircraft engines (1945)



- an error causing unintended behavior

- an error causing unintended behavior
- software vs. hardware bugs

- an error causing unintended behavior
- software vs. hardware bugs
- bugs vs. backdoors

- an error causing unintended behavior
- software vs. hardware bugs
- bugs vs. backdoors
- bugs vs. specification flaws

- Human Expert
- Static Analysis
- Sanitizer
- Debugging

- Human Expert
- Static Analysis
- Sanitizer
- Debugging

Access to Source/Binary?

Human Expert



**HUMAN
EXPERT**





- find bugs that occur very rarely (not found in automated testing)



- find bugs that occur very rarely (not found in automated testing)
- find flaws which might be exploitable with future code changes



- find bugs that occur very rarely (not found in automated testing)
- find flaws which might be exploitable with future code changes
- develop better practices



- read code, try to find flaws



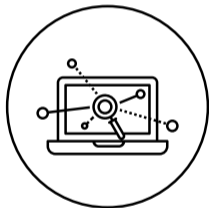
- read code, try to find flaws
- common recommendation: peer-review code when committing it



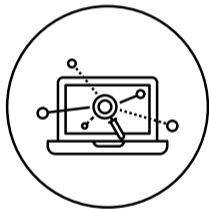
- slow and expensive: you might need as many reviewers as developers



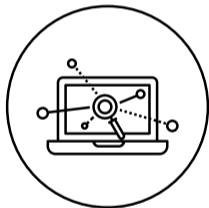
- slow and expensive: you might need as many reviewers as developers
- humans are error-prone and bugs remain



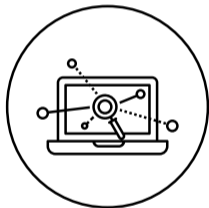
- complement code review



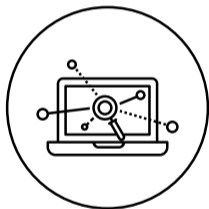
- complement code review
- identify, quantify, and address security risks



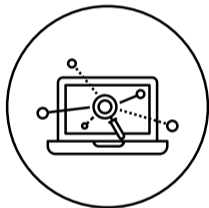
- complement code review
- identify, quantify, and address security risks
- three step process



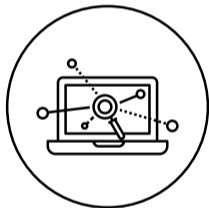
- complement code review
- identify, quantify, and address security risks
- three step process
 1. Decompose



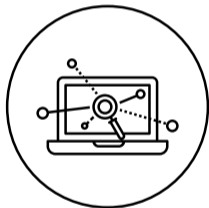
- complement code review
- identify, quantify, and address security risks
- three step process
 1. Decompose
 2. Determine threats



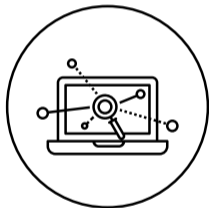
- complement code review
- identify, quantify, and address security risks
- three step process
 1. Decompose
 2. Determine threats
 3. Determine mitigations



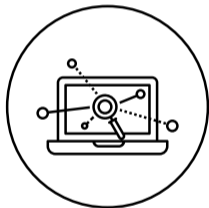
- create use cases



- create use cases
- model interaction with an attacker

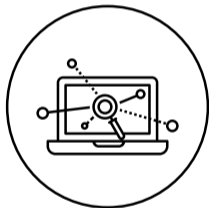


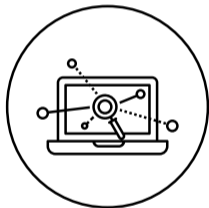
- create use cases
- model interaction with an attacker
- model assets (target data)



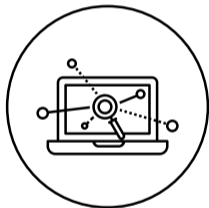
- create use cases
- model interaction with an attacker
- model assets (target data)
- strictly defined procedure, to avoid incorrect or incomplete deductions

- Use a threat categorization technique (STRIDE/ASF)

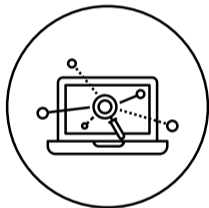




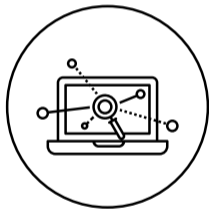
- Use a threat categorization technique (STRIDE/ASF)
- Various models to put a number on a potential attack



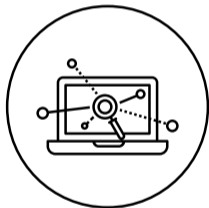
- Use a threat categorization technique (STRIDE/ASF)
- Various models to put a number on a potential attack
- DREAD Score (Microsoft) averages over:



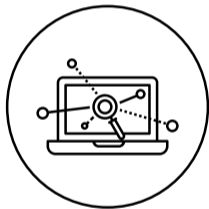
- Use a threat categorization technique (STRIDE/ASF)
- Various models to put a number on a potential attack
- DREAD Score (Microsoft) averages over:
 1. Damage potential



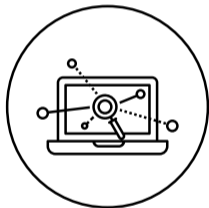
- Use a threat categorization technique (STRIDE/ASF)
- Various models to put a number on a potential attack
- DREAD Score (Microsoft) averages over:
 1. Damage potential
 2. Reproducibility



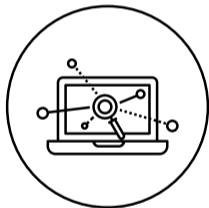
- Use a threat categorization technique (STRIDE/ASF)
- Various models to put a number on a potential attack
- DREAD Score (Microsoft) averages over:
 1. Damage potential
 2. Reproducibility
 3. Exploitability



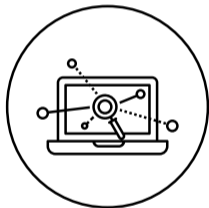
- Use a threat categorization technique (STRIDE/ASF)
- Various models to put a number on a potential attack
- DREAD Score (Microsoft) averages over:
 1. Damage potential
 2. Reproducibility
 3. Exploitability
 4. Affected users



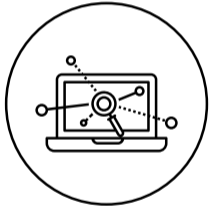
- Use a threat categorization technique (STRIDE/ASF)
- Various models to put a number on a potential attack
- DREAD Score (Microsoft) averages over:
 1. Damage potential
 2. Reproducibility
 3. Exploitability
 4. Affected users
 5. Discoverability



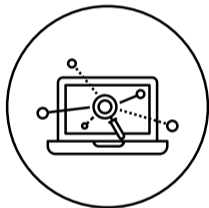
- For every potential attack there must be a mitigation



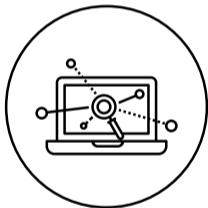
- For every potential attack there must be a mitigation
- If you figure out that there is no mitigation:



- For every potential attack there must be a mitigation
- If you figure out that there is no mitigation:
 - that's a problem

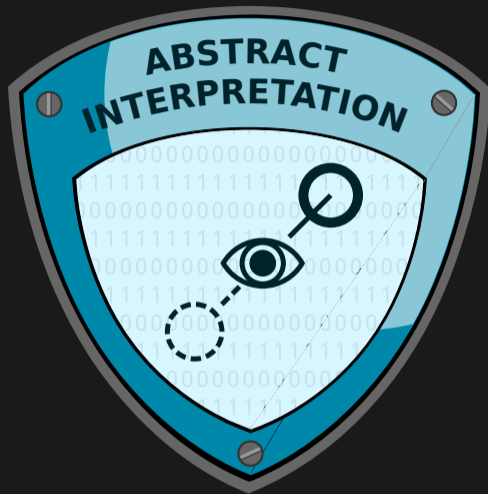


- For every potential attack there must be a mitigation
- If you figure out that there is no mitigation:
 - that's a problem
 - but it's good that you found it



- For every potential attack there must be a mitigation
- If you figure out that there is no mitigation:
 - that's a problem
 - but it's good that you found it
 - you already know how significant the problem is

Static Analysis



- Can we (automatically) find bugs without even executing the program?

- Can we (automatically) find bugs without even executing the program?
- Execute an abstracted variant of the program (maintaining semantics)

```
// randomly select and return one argument  
int main(int argc, char* argv[]) {  
    int max = argc;  
    int index = rand() % max;  
    return argv[index];  
}
```

- $\text{argc} \in [0, +\infty]$

```
// randomly select and return one argument  
int main(int argc, char* argv[]) {  
    int max = argc;  
    int index = rand() % max;  
    return argv[index];  
}
```

- $argc \in [0, +\infty]$
- $max \in [0, +\infty]$

```
// randomly select and return one argument  
int main(int argc, char* argv[]) {  
    int max = argc;  
    int index = rand() % max;  
    return argv[index];  
}
```

- $argc \in [0, +\infty]$
- $max \in [0, +\infty]$
- $rand() \% max \rightarrow$ possible division by zero

- Formalized to allow sound approximation

- Formalized to allow sound approximation
- Symbolic execution

- Formalized to allow sound approximation
- Symbolic execution
- Basis of Static Analysis

STATIC ANALYSIS



Works on source code (could also be applied to binary code)

Works on source code (could also be applied to binary code)

- Examine every possible code path (→ symbolic execution)

Works on source code (could also be applied to binary code)

- Examine every possible code path (→ symbolic execution)
- Consider every possible input (→ symbolic execution)

Works on source code (could also be applied to binary code)

- Examine every possible code path (→ symbolic execution)
- Consider every possible input (→ symbolic execution)

→ Good coverage

Works on source code (could also be applied to binary code)

- Examine every possible code path (→ symbolic execution)
- Consider every possible input (→ symbolic execution)

→ Good coverage

- Should be run continuously during development

Works on source code (could also be applied to binary code)

- Examine every possible code path (→ symbolic execution)
- Consider every possible input (→ symbolic execution)

→ Good coverage

- Should be run continuously during development
- Can also be run afterwards to find bugs

- Can Static Code Analysis find all problems?

- Can Static Code Analysis find all problems?
 - Equivalent to solving the halting problem

- Can Static Code Analysis find all problems?
 - Equivalent to solving the halting problem
 - There are false negatives!

- Can Static Code Analysis find all problems?
 - Equivalent to solving the halting problem
 - There are false negatives!
- Can we have false positives?

- Can Static Code Analysis find all problems?
 - Equivalent to solving the halting problem
 - There are false negatives!
- Can we have false positives?
 - Yes, if the assumptions the static code analyzer makes are too loose

- Can Static Code Analysis find all problems?
 - Equivalent to solving the halting problem
 - There are false negatives!
- Can we have false positives?
 - Yes, if the assumptions the static code analyzer makes are too loose
 - e.g., a check/bound is not evident in the source code or for the analyzer

- Popular static code analyzer

- Popular static code analyzer
- `scan-build make`

- Popular static code analyzer
- `scan-build make`
- Generates an error report

```
83 int main(int argc, char **argv) {  
84     if (argc != 3) {
```

1 Assuming 'argc' is equal to 3 →

2 ← Taking false branch →

```
85         fprintf(stderr, "usage: %s elf-file dbg-file\n", argv[0]);  
86         return 2;  
87     }
```

```
89     int fd = open(argv[1], O_RDONLY);  
90     if (fd < 0) {
```

3 ← Assuming 'fd' is ≥ 0 →

4 ← Taking false branch →

```
91         fprintf(stderr, "%s: %s\n", argv[1], strerror(errno));  
92         return 0;  
93     }
```

```
94 FILE *d = fopen(argv[2], "wb");  
95 if (!d) return 0;
```

5 ← Assuming 'd' is non-null →

6 ← Taking false branch →

```
96  
97 char *debug = (char *) malloc(1024 * 1024);
```

7 ← Memory is allocated →

242

```
fclose(d);
```

8 ← Potential leak of memory pointed to by 'debug_start'

243

```
return 0;
```

244

```
}
```

```
277 int fd;
    5 ← 'fd' declared without an initial value →
278
279 printf ("Size limit = %lld bytes\n", opts.sizelimit);
280
281 if (opts.hugetlb)
    6 ← Assuming the condition is false →
    7 ← Taking false branch →
282 {
```

```
318     if (opts.hugetlb)
319     {
320         munmap (array, opts.sizelimit);
321         unlink ("/mnt/test/tempfile");
322         close (fd);
323     }
```

17 ← Assuming the condition is true →

18 ← Taking true branch →

19 ← Function call argument is an uninitialized value

Sanitizers



Address Sanitizer (ASan)...

Address Sanitizer (ASan)...

- is a compiler extension implemented in GCC and Clang
- is activated with compiler flag `-fsanitize=address`

Address Sanitizer (ASan)...

- is a compiler extension implemented in GCC and Clang
- is activated with compiler flag `-fsanitize=address`
- instruments code to detect memory errors at runtime
- should only be used as debugging tool





ASan detects



ASan detects

- Out-of-bounds accesses to heap, stack and globals



ASan detects

- Out-of-bounds accesses to heap, stack and globals
- Use-after-free
- Use-after-return
- Use-after-scope



ASan detects

- Out-of-bounds accesses to heap, stack and globals
- Use-after-free
- Use-after-return
- Use-after-scope
- Double-free, invalid free



ASan detects

- Out-of-bounds accesses to heap, stack and globals
- Use-after-free
- Use-after-return
- Use-after-scope
- Double-free, invalid free
- Memory leaks (experimental)

```
typedef struct {
    void (*print)(char*);
} operation;

int main(int argc, char* argv[]) {
    operation* io = (operation*)malloc(sizeof(operation));
    io->print = puts;
    io->print("Hallo ");
    free(io);

    if(argc > 1) {
        char* buffer = (char*)malloc(8);
        strncpy(buffer, argv[1], 7);
        io->print(buffer);
        free(buffer);
    }
    return 0;
}
```

Hallo

```
=====
==27728==ERROR: AddressSanitizer: heap-use-after-free on address 0x60200000eff0 at pc 0x00000040094d bp 0x7fffffffcd0 sp 0x7fffffffdcc0
READ of size 8 at 0x60200000eff0 thread T0
#0 0x40094c in main /home/mschwarz/Teaching/IIS/asan.c:24
#1 0x7ffff6ac182f in __libc_start_main (/lib/x86_64-linux-gnu/libc.so.6+0x2082f)
#2 0x4007f8 in _start (/home/mschwarz/Teaching/IIS/a.out+0x4007f8)
```

```
0x60200000eff0 is located 0 bytes inside of 8-byte region [0x60200000eff0,0x60200000eff8)
freed by thread T0 here:
#0 0x7ffff6f022ca in __interceptor_free (/usr/lib/x86_64-linux-gnu/libasan.so.2+0x982ca)
#1 0x4008f7 in main /home/mschwarz/Teaching/IIS/asan.c:19
#2 0x7ffff6ac182f in __libc_start_main (/lib/x86_64-linux-gnu/libc.so.6+0x2082f)
```

```
previously allocated by thread T0 here:
#0 0x7ffff6f02602 in malloc (/usr/lib/x86_64-linux-gnu/libasan.so.2+0x98602)
#1 0x4008e2 in main /home/mschwarz/Teaching/IIS/asan.c:16
#2 0x7ffff6ac182f in __libc_start_main (/lib/x86_64-linux-gnu/libc.so.6+0x2082f)
```

SUMMARY: AddressSanitizer: heap-use-after-free /home/mschwarz/Teaching/IIS/asan.c:24 main

Shadow bytes around the buggy address:

```
0x0c047fff9da0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
0x0c047fff9db0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
0x0c047fff9dc0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
0x0c047fff9dd0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
0x0c047fff9de0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
=>0x0c047fff9df0: fa fa fa fa fa fa fa fa fa fa 00 fa fa fa [fd]fa
0x0c047fff9e00: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
0x0c047fff9e10: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
0x0c047fff9e20: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
0x0c047fff9e30: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
0x0c047fff9e40: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
```

Shadow byte legend (one shadow byte represents 8 application bytes):

```
Addressable: 00
Partially addressable: 01 02 03 04 05 06 07
Heap left redzone: fa
Heap right redzone: fb
Freed heap region: fd
Stack left redzone: f1
Stack mid redzone: f2
Stack right redzone: f3
Stack partial redzone: f4
Stack after return: f5
Stack use after scope: f8
Global redzone: f9
Global init order: f6
Poisoned by user: f7
Container overflow: fc
Array cookie: ac
Intra object redzone: bb
ASan internal: fe
```

==27728==ABORTING

Address sanitizer introduces shadow memory:

Address sanitizer introduces shadow memory:

- Allocate 1 shadow byte per 8-byte word

Address sanitizer introduces shadow memory:

- Allocate 1 shadow byte per 8-byte word
- The shadow byte tracks these 9 states:

Address sanitizer introduces shadow memory:

- Allocate 1 shadow byte per 8-byte word
- The shadow byte tracks these 9 states:
 - `00` → all 8 bytes are addressable
 - `01-07` → first 1-7 bytes are addressable
 - `80-ff` → none of the bytes are addressable

Address sanitizer introduces shadow memory:

- Allocate 1 shadow byte per 8-byte word
- The shadow byte tracks these 9 states:
 - `00` → all 8 bytes are addressable
 - `01-07` → first 1-7 bytes are addressable
 - `80-ff` → none of the bytes are addressable
- Check the shadow memory on each access

Address sanitizer introduces shadow memory:

- Allocate 1 shadow byte per 8-byte word
- The shadow byte tracks these 9 states:
 - `00` → all 8 bytes are addressable
 - `01-07` → first 1-7 bytes are addressable
 - `80-ff` → none of the bytes are addressable
- Check the shadow memory on each access
- Allocate and update shadow memory as needed

Address sanitizer introduces shadow memory:

- Allocate 1 shadow byte per 8-byte word
- The shadow byte tracks these 9 states:
 - `00` → all 8 bytes are addressable
 - `01-07` → first 1-7 bytes are addressable
 - `80-ff` → none of the bytes are addressable
- Check the shadow memory on each access
- Allocate and update shadow memory as needed

- Here is an example

```
#include <stdlib.h>
int main(int argc, char* argv[])
{
    // argc is 1
    int* a = malloc(22);
    a[0*argc] = 5;
    a[4*argc] = 42;
    a[3*argc] = 15;
    a[5*argc] = 18;
    free(a);
    return 0;
}
// Shadow memory:
// fa fa fa fa fa fa fa fa
// fa fa fa fa fa fa fa fa
// fa fa fa fa fa fa fa fa
```

Address sanitizer introduces shadow memory:

- Allocate 1 shadow byte per 8-byte word
- The shadow byte tracks these 9 states:
 - `00` → all 8 bytes are addressable
 - `01-07` → first 1-7 bytes are addressable
 - `80-ff` → none of the bytes are addressable
- Check the shadow memory on each access
- Allocate and update shadow memory as needed

- Here is an example

```
#include <stdlib.h>
int main(int argc, char* argv[])
{
    // argc is 1
    int* a = malloc(22); // <--
    a[0*argc] = 5;
    a[4*argc] = 42;
    a[3*argc] = 15;
    a[5*argc] = 18;
    free(a);
    return 0;
}
// Shadow memory:
// fa fa fa fa fa fa fa fa
// fa fa fa fa[00 00 06]fa
// fa fa fa fa fa fa fa fa
```

Address sanitizer introduces shadow memory:

- Allocate 1 shadow byte per 8-byte word
- The shadow byte tracks these 9 states:
 - 00 → all 8 bytes are addressable
 - 01-07 → first 1-7 bytes are addressable
 - 80-ff → none of the bytes are addressable
- Check the shadow memory on each access
- Allocate and update shadow memory as needed

- Here is an example

```
#include <stdlib.h>
int main(int argc, char* argv[])
{
    // argc is 1
    int* a = malloc(22);
    a[0*argc] = 5; // <--
    a[4*argc] = 42;
    a[3*argc] = 15;
    a[5*argc] = 18;
    free(a);
    return 0;
}
// Shadow memory:
// fa fa fa fa fa fa fa fa
// fa fa fa fa[00]00 06 fa
// fa fa fa fa fa fa fa fa
```

Address sanitizer introduces shadow memory:

- Allocate 1 shadow byte per 8-byte word
- The shadow byte tracks these 9 states:
 - 00 → all 8 bytes are addressable
 - 01-07 → first 1-7 bytes are addressable
 - 80-ff → none of the bytes are addressable
- Check the shadow memory on each access
- Allocate and update shadow memory as needed

- Here is an example

```
#include <stdlib.h>
int main(int argc, char* argv[])
{
    // argc is 1
    int* a = malloc(22);
    a[0*argc] = 5;
    a[4*argc] = 42; // <--
    a[3*argc] = 15;
    a[5*argc] = 18;
    free(a);
    return 0;
}
// Shadow memory:
// fa fa fa fa fa fa fa fa
// fa fa fa fa 00 00[06]fa
// fa fa fa fa fa fa fa fa
```

Address sanitizer introduces shadow memory:

- Allocate 1 shadow byte per 8-byte word
- The shadow byte tracks these 9 states:
 - 00 → all 8 bytes are addressable
 - 01-07 → first 1-7 bytes are addressable
 - 80-ff → none of the bytes are addressable
- Check the shadow memory on each access
- Allocate and update shadow memory as needed

- Here is an example

```
#include <stdlib.h>
int main(int argc, char* argv[])
{
    // argc is 1
    int* a = malloc(22);
    a[0*argc] = 5;
    a[4*argc] = 42;
    a[3*argc] = 15; // <--
    a[5*argc] = 18;
    free(a);
    return 0;
}
// Shadow memory:
// fa fa fa fa fa fa fa fa
// fa fa fa fa 00[00]06 fa
// fa fa fa fa fa fa fa fa
```

Address sanitizer introduces shadow memory:

- Allocate 1 shadow byte per 8-byte word
- The shadow byte tracks these 9 states:
 - 00 → all 8 bytes are addressable
 - 01-07 → first 1-7 bytes are addressable
 - 80-ff → none of the bytes are addressable
- Check the shadow memory on each access
- Allocate and update shadow memory as needed

- Here is an example

```
#include <stdlib.h>
int main(int argc, char* argv[])
{
    // argc is 1
    int* a = malloc(22);
    a[0*argc] = 5;
    a[4*argc] = 42;
    a[3*argc] = 15;
    a[5*argc] = 18; // <-- ERROR
    free(a);
    return 0;
}
// Shadow memory:
// fa fa fa fa fa fa fa fa
// fa fa fa fa 00 00[06]fa
// fa fa fa fa fa fa fa fa
```

Address sanitizer introduces shadow memory:

- Allocate 1 shadow byte per 8-byte word
- The shadow byte tracks these 9 states:
 - 00 → all 8 bytes are addressable
 - 01-07 → first 1-7 bytes are addressable
 - 80-ff → none of the bytes are addressable
- Check the shadow memory on each access
- Allocate and update shadow memory as needed

- Here is an example

```
#include <stdlib.h>
int main(int argc, char* argv[])
{
    // argc is 1
    int* a = malloc(22);
    a[0*argc] = 5;
    a[4*argc] = 42;
    a[3*argc] = 15;
    a[5*argc] = 18;
    free(a);          // <--
    return 0;
}
// Shadow memory:
// fa fa fa fa fa fa fa fa
// fa fa fa fa[fd fd fd]fa
// fa fa fa fa fa fa fa fa
```


Address sanitizer introduces shadow memory:

- Allocate 1 shadow byte per 8-byte word
- The shadow byte tracks these 9 states:
 - `00` → all 8 bytes are addressable
 - `01-07` → first 1-7 bytes are addressable
 - `80-ff` → none of the bytes are addressable
- Check the shadow memory on each access
- Allocate and update shadow memory as needed

- Here is an example

```
#include <stdlib.h>
int main(int argc, char* argv[])
{
    // argc is 1
    int* a = malloc(22);
    a[0*argc] = 5;
    a[4*argc] = 42;
    a[3*argc] = 15;
    a[5*argc] = 18;
    free(a);
    return 0;          // <--
}
// Shadow memory:
// fa fa fa fa fa fa fa fa
// fa fa fa fa fd fd fd fa
// fa fa fa fa fa fa fa fa
```

- `malloc` and `free` are replaced



- `malloc` and `free` are replaced
- Memory around `malloc`'d segments is poisoned

```
// ff ff fa fa fa 00 00 00 04 fa fa fa ff ff  
//      [red zone] allocated [red zone]
```





- malloc and free are replaced
- Memory around malloc'd segments is poisoned
- free'd memory is poisoned and moved to quarantine

```
// ff ff fa fa fa 00 00 00 04 fa fa fa ff ff  
//      [red zone] allocated [red zone]
```

```
// ff ff fa fa fa fd fd fd fd fa fa fa ff ff  
//      red zone[ quarantine]red zone
```



- malloc and free are replaced
- Memory around malloc'd segments is poisoned
- free'd memory is poisoned and moved to quarantine

```
// ff ff fa fa fa 00 00 00 04 fa fa fa ff ff  
//      [red zone] allocated [red zone]
```

```
// ff ff fa fa fa fd fd fd fd fa fa fa ff ff  
//      red zone[ quarantine]red zone
```

- Every memory access is first checked if poisoned

```
if (IsPoisoned(address)) {  
    ReportError(address, kAccessSize, kIsWrite);  
}  
*address = ...; // or: ... = *address;
```



- malloc and free are replaced
- Memory around malloc'd segments is poisoned

```
// ff ff fa fa fa 00 00 00 04 fa fa fa ff ff  
//      [red zone] allocated [red zone]
```

- free'd memory is poisoned and moved to quarantine

```
// ff ff fa fa fa fd fd fd fd fa fa fa ff ff  
//      red zone[ quarantine]red zone
```

- Every memory access is first checked if poisoned

```
if (IsPoisoned(address)) {  
    ReportError(address, kAccessSize, kIsWrite);  
}  
*address = ...; // or: ... = *address;
```

- red zones are also added to stack allocation and global arrays

Limitations

- Slowdown of approximately factor 2
- Increased memory usage of factor 2 to 5, depending on allocations
- Cannot prevent all arbitrary memory corruptions
- Does not protect against integer overflows
- Adjacent buffers in structs/classes are not protected

Leak Sanitizer ...

- part of ASan, but also available as standalone



Leak Sanitizer ...

- part of ASan, but also available as standalone
- implemented in GCC and Clang
- is activated with compiler flag `-fsanitize=leak`



Leak Sanitizer ...

- part of ASan, but also available as standalone
- implemented in GCC and Clang
- is activated with compiler flag `-fsanitize=leak`
- detects memory leaks at runtime



Leak Sanitizer ...

- part of ASan, but also available as standalone
- implemented in GCC and Clang
- is activated with compiler flag `-fsanitize=leak`
- detects memory leaks at runtime
- should only be used as debugging tool



Leak Sanitizer ...

- part of ASan, but also available as standalone



Leak Sanitizer ...

- part of ASan, but also available as standalone
- implemented in GCC and Clang
- is activated with compiler flag `-fsanitize=leak`



Leak Sanitizer ...

- part of ASan, but also available as standalone
- implemented in GCC and Clang
- is activated with compiler flag `-fsanitize=leak`
- detects memory leaks at runtime



Leak Sanitizer ...

- part of ASan, but also available as standalone
- implemented in GCC and Clang
- is activated with compiler flag `-fsanitize=leak`
- detects memory leaks at runtime
- should only be used as debugging tool



```
char* buffer;
void print(const char* tag, const char* info)
{
    buffer = malloc(strlen(tag) + strlen(info) + 4);
    strcpy(buffer, tag);
    strcat(buffer, ": ");
    strcat(buffer, info);
    strcat(buffer, "\n");
    puts(buffer);
    free(buffer);
}
int main(int argc, char* argv[]) {
    buffer = malloc(32);
    fgets(buffer, 32, stdin);
    const char* tag = strdup("user input");
    print(tag, buffer);
    return 0;
}
```



```
hello world
user input: hello world
```

```
=====  
==20366==ERROR: LeakSanitizer: detected memory leaks
```

```
Direct leak of 32 byte(s) in 1 object(s) allocated from:
```

```
 #0 0x7f3a3bfb8fee in __interceptor_malloc (/usr/lib/x86_64-linux-gnu/liblsan.  
 #1 0x559f9420fa80 in main /home/dgruss/buffer.c:17  
 #2 0x7f3a3bc053f0 in __libc_start_main (/lib/x86_64-linux-gnu/libc.so.6+0x203
```

```
Direct leak of 11 byte(s) in 1 object(s) allocated from:
```

```
 #0 0x7f3a3bfb8fee in __interceptor_malloc (/usr/lib/x86_64-linux-gnu/liblsan.  
 #1 0x7f3a3bc72799 in __strdup (/lib/x86_64-linux-gnu/libc.so.6+0x8d799)
```

```
SUMMARY: LeakSanitizer: 43 byte(s) leaked in 2 allocation(s).
```

- Rewrite code, add calls to read/write access check function

- Rewrite code, add calls to read/write access check function
- Performance overhead: 2 – 20×

- Rewrite code, add calls to read/write access check function
- Performance overhead: 2 – 20×
- Memory impact: 5 – 10×

Store in shadow memory for each 8-byte region:

Store in shadow memory for each 8-byte region:

- TID (16 bit)
- Timestamp (42 bit)
- Position/Size in 8-byte word (5 bit)
- Read/Write (1 bit)

Store in shadow memory for each 8-byte region:

- TID (16 bit)
- Timestamp (42 bit)
- Position/Size in 8-byte word (5 bit)
- Read/Write (1 bit)

→ 8 bytes

Store in shadow memory for each 8-byte region:

- TID (16 bit)
- Timestamp (42 bit)
- Position/Size in 8-byte word (5 bit)
- Read/Write (1 bit)

→ 8 bytes

Buffer the last $N \in [4, 9]$ memory accesses

Store in shadow memory for each 8-byte region:

- TID (16 bit)
- Timestamp (42 bit)
- Position/Size in 8-byte word (5 bit)
- Read/Write (1 bit)

→ 8 bytes

Buffer the last $N \in [4, 9]$ memory accesses

- per 8-byte region
- Memory impact: 5 – 10×

```
void enter_zoo(size_t animal) {
    lock_chamberlock1(animal);
    lock_chamberlock2(animal);
    printf("Waiting for animal %zd to wake up.\n", animal);
    NANOSLEEP(1000L* 1000L);
    printf("Reopen transfer chamber to place animal %zd in zoo.\n", animal);
    unlock_chamberlock2(animal);
    unlock_chamberlock1(animal);
}

void leave_zoo(size_t animal) {
    lock_chamberlock2(animal);
    lock_chamberlock1(animal);
    printf("Waiting for animal %zd to sleep.\n", animal);
    NANOSLEEP(1000L* 1000L);
    printf("Reopen transfer chamber to remove animal %zd from the zoo.\n", animal);
    unlock_chamberlock1(animal);
    unlock_chamberlock2(animal);
}
```

```
=====
WARNING: ThreadSanitizer: lock-order-inversion (potential deadlock) (pid=26054)
  Cycle in lock order graph: M0 (0x559e51fa7040) => M1 (0x559e51fa7080) => M0

  Mutex M1 acquired here while holding mutex M0 in thread T1:
    #0 pthread_mutex_lock <null> (libtsan.so.0+0x00000003b8ee)
    #1 lock_chamberlock2 /tmp/snp2018g289/A6/animal_transport/animal_transport.c:38 (animal_transport+0x0000000010c5)
    #2 enter_zoo /tmp/snp2018g289/A6/animal_transport/animal_transport.c:51 (animal_transport+0x000000001168)
    #3 animal_do /tmp/snp2018g289/A6/animal_transport/animal_transport.c:72 (animal_transport+0x0000000012e5)
    #4 <null> <null> (libtsan.so.0+0x000000025aab)

  Hint: use TSAN_OPTIONS=second_deadlock_stack=1 to get more informative warning message

  Mutex M0 acquired here while holding mutex M1 in thread T1:
    #0 pthread_mutex_lock <null> (libtsan.so.0+0x00000003b8ee)
    #1 lock_chamberlock1 /tmp/snp2018g289/A6/animal_transport/animal_transport.c:26 (animal_transport+0x00000000103d)
    #2 leave_zoo /tmp/snp2018g289/A6/animal_transport/animal_transport.c:62 (animal_transport+0x00000000122c)
    #3 animal_do /tmp/snp2018g289/A6/animal_transport/animal_transport.c:77 (animal_transport+0x00000000137a)
    #4 <null> <null> (libtsan.so.0+0x000000025aab)

  Thread T1 (tid=26056, running) created by main thread at:
    #0 pthread_create <null> (libtsan.so.0+0x0000000290c3)
    #1 main /tmp/snp2018g289/A6/animal_transport/animal_transport.c:110 (animal_transport+0x000000001517)

SUMMARY: ThreadSanitizer: lock-order-inversion (potential deadlock) (/usr/lib/x86_64-linux-gnu/libtsan.so.0+0x3b8ee) in __interceptor_pthread_mutex_lock
=====
```

Memory sanitizer (MSan) ...

- is not part of ASan, but similar interface

Memory sanitizer (MSan) ...

- is not part of ASan, but similar interface
- detects use of uninitialized values

Memory sanitizer (MSan) ...

- is not part of ASan, but similar interface
- detects use of uninitialized values
- implemented in GCC and Clang
- is activated with compiler flag `-fsanitize=memory`

Memory sanitizer (MSan) ...

- is not part of ASan, but similar interface
- detects use of uninitialized values
- implemented in GCC and Clang
- is activated with compiler flag `-fsanitize=memory`
- aims at replacing valgrind memcheck tool

Memory sanitizer (MSan) ...

- is not part of ASan, but similar interface
- detects use of uninitialized values
- implemented in GCC and Clang
- is activated with compiler flag `-fsanitize=memory`
- aims at replacing valgrind memcheck tool

- Implements a bit precise shadow memory tracking initialization
- Each operation on normal memory has similar operation:
 - $A = B \& C \rightarrow A' = (B' \& C') | (B \& C')$
 - $A = B | C \rightarrow A' = (B' \& C') | (\sim B \& C')$
 - ...
- Copying and use of undefined memory are generally allowed
- Reports a bug whenever undefined value influences control flow
- Typical slowdown: 3×

- Implements a bit precise shadow memory tracking initialization
- Each operation on normal memory has similar operation:
 - $A = B \& C \rightarrow A' = (B' \& C') | (B \& C')$
 - $A = B | C \rightarrow A' = (B' \& C') | (\sim B \& C')$
 - ...
- Copying and use of undefined memory are generally allowed
- Reports a bug whenever undefined value influences control flow
- Typical slowdown: $3\times$
- Memory impact: $2 - 3\times$

- Implements a bit precise shadow memory tracking initialization
- Each operation on normal memory has similar operation:
 - $A = B \& C \rightarrow A' = (B' \& C') \mid (B \& C')$
 - $A = B \mid C \rightarrow A' = (B' \& C') \mid (\sim B \& C')$
 - ...
- Copying and use of undefined memory are generally allowed
- Reports a bug whenever undefined value influences control flow
- Typical slowdown: $3\times$
- Memory impact: $2 - 3\times$
- reserves a 64 TB region virtual address space for shadows

```
int main(int argc, char** argv) {  
    int* a = new int[10];  
    a[5] = 0;  
    volatile int b = a[argc];  
    if (b)  
        printf("xx\n");  
    return 0;  
}
```

```
==25491==WARNING: MemorySanitizer: use-of-uninitialized-value
```

```
#0 0x563fa6533efe (/home/dgruss/a.out+0x93efe)
```

```
#1 0x7fd1cbbd83f0 (/lib/x86_64-linux-gnu/libc.so.6+0x203f0)
```

```
#2 0x563fa64baff9 (/home/dgruss/a.out+0x1aff9)
```

```
Uninitialized value was stored to memory at
```

```
#0 0x563fa6533e73 (/home/dgruss/a.out+0x93e73)
```

```
#1 0x7fd1cbbd83f0 (/lib/x86_64-linux-gnu/libc.so.6+0x203f0)
```

```
Uninitialized value was created by a heap allocation
```

```
#0 0x563fa65312a0 (/home/dgruss/a.out+0x912a0)
```

```
#1 0x563fa6533c58 (/home/dgruss/a.out+0x93c58)
```

```
#2 0x7fd1cbbd83f0 (/lib/x86_64-linux-gnu/libc.so.6+0x203f0)
```

```
SUMMARY: MemorySanitizer: use-of-uninitialized-value (/home/dgruss/a.o
```

```
Exiting
```



Debugging

- How does a debugger work?

- How does a debugger work?
- Let's build a debugger!

- How does a debugger work?
- Let's build a debugger!
- What do we need?

- A way to read and write basically any register and memory location of another process

- A way to read and write basically any register and memory location of another process
→ `ptrace()`

- A way to read and write basically any register and memory location of another process
→ `ptrace()`
- A way to interrupt a process at an arbitrary instruction

- A way to read and write basically any register and memory location of another process
→ `ptrace()`
- A way to interrupt a process at an arbitrary instruction
→ `interrupts interrupt processes`

```
#include <sys/ptrace.h>
long ptrace(enum __ptrace_request request, pid_t pid, void *addr, void *data);
```

```
#include <sys/ptrace.h>  
long ptrace(enum __ptrace_request request, pid_t pid, void *addr, void *data);
```

ptrace(): one process observes and controls the execution of another process, examine and change the its memory, and registers.


```
#include <sys/ptrace.h>
long ptrace(enum __ptrace_request request, pid_t pid, void *addr, void *data);
```

ptrace(): one process observes and controls the execution of another process, examine and change the its memory, and registers.

- primarily used for breakpoint debugging and system call tracing

```
#include <sys/ptrace.h>  
long ptrace(enum __ptrace_request request, pid_t pid, void *addr, void *data);
```

`ptrace()`: one process observes and controls the execution of another process, examine and change the its memory, and registers.

- primarily used for breakpoint debugging and system call tracing
- permission system:

```
#include <sys/ptrace.h>
long ptrace(enum __ptrace_request request, pid_t pid, void *addr, void *data);
```

`ptrace()`: one process observes and controls the execution of another process, examine and change the its memory, and registers.

- primarily used for breakpoint debugging and system call tracing
- permission system:
 - trace other threads of your own process

```
#include <sys/ptrace.h>
long ptrace(enum __ptrace_request request, pid_t pid, void *addr, void *data);
```

`ptrace()`: one process observes and controls the execution of another process, examine and change the its memory, and registers.

- primarily used for breakpoint debugging and system call tracing
- permission system:
 - trace other threads of your own process
 - trace child processes

```
#include <sys/ptrace.h>
long ptrace(enum __ptrace_request request, pid_t pid, void *addr, void *data);
```

`ptrace()`: one process observes and controls the execution of another process, examine and change the its memory, and registers.

- primarily used for breakpoint debugging and system call tracing
- permission system:
 - trace other threads of your own process
 - trace child processes
 - trace any process if you're root (`CAP_SYS_PTRACE`)

```
#include <sys/ptrace.h>
long ptrace(enum __ptrace_request request, pid_t pid, void *addr, void *data);
```

`ptrace()`: one process observes and controls the execution of another process, examine and change the its memory, and registers.

- primarily used for breakpoint debugging and system call tracing
- permission system:
 - trace other threads of your own process
 - trace child processes
 - trace any process if you're root (`CAP_SYS_PTRACE`)
- check `ptrace()` man page!

- read and write memory: `PTRACE_PEEKTEXT`, `PTRACE_PEEKUSER`, ...
- read and write registers: `PTRACE_GETREGSET`, `PTRACE_SETREGS`
- monitor syscalls: `PTRACE_SYSCALL`, ...
 - receive a `SIGTRAP` upon specific or all system calls
- several others

1. fork
 2. child calls `ptrace (PTTRACE_TRACEME, 0, 0, 0);`
 3. parent calls `ptrace (PTTRACE_ATTACH, child_pid, 0, 0);`
- parent now controls the child

- Operating system provides special interrupt handling for debuggers
- Software interrupt 3
- How can we use that?

```
int main(int argc, char* argv[]) {
    if (argc <= 1)
        printf("usage: ./tohex <number> <base>\n");
    unsigned int base = 0;
    for (ssize_t i = 0; i < strlen(argv[2]); ++i) {
        base *= 10;
        base = argv[2][i] - '0';
    }
    printf("base = %u\n",base);
    unsigned int result = 0;
    for (ssize_t i = 0; i < strlen(argv[1]); ++i) {
        result *= 10;
        if (argv[1][i] >= '0' && argv[1][i] <= '9')
            result = argv[1][i] - '0';
        if (argv[1][i] >= 'A' && argv[1][i] <= 'Z')
            result = argv[1][i] - 'A';
        if (argv[1][i] >= 'a' && argv[1][i] <= 'z')
            result = argv[1][i] - 'a';
    }
    printf("%u\n", result);
}
```

```
6da: 55                push   %rbp
6db: 48 89 e5          mov    %rsp,%rbp
6de: 48 83 ec 30       sub    $0x30,%rsp
6e2: 89 7d dc          mov    %edi,-0x24(%rbp)
6e5: 48 89 75 d0       mov    %rsi,-0x30(%rbp)
6e9: 83 7d dc 01       cmpl  $0x1,-0x24(%rbp)
6ed: 7f 0c            jg     6fb <main+0x21>
6ef: 48 8d 3d 62 02 00 00 lea   0x262(%rip),%rdi
6f6: e8 95 fe ff ff    callq 590 <puts@plt>
6fb: c7 45 e8 00 00 00 00 movl  $0x0,-0x18(%rbp)
702: 48 c7 45 f0 00 00 00 movq  $0x0,-0x10(%rbp)
```

```
6da: 55          push   %rbp
6db: 48 89 e5    mov    %rsp,%rbp
6de: 48 83 ec 30 sub    $0x30,%rsp
6e2: 89 7d dc    mov    %edi,-0x24(%rbp)
6e5: 48 89 75 d0 mov    %rsi,-0x30(%rbp)
6e9: 83 7d dc 01 cmpl   $0x1,-0x24(%rbp)
6ed: 7f 0c      jg     6fb <main+0x21>
6ef: 48 8d 3d 62 02 00 00 lea   0x262(%rip),%rdi
6f6: e8 95 fe ff ff callq  590 <puts@plt>
6fb: c7 45 e8 00 00 00 00 movl  $0x0,-0x18(%rbp)
702: 48 c7 45 f0 00 00 00 movq  $0x0,-0x10(%rbp)
```

How can we interrupt a program here?

Idea: Just replace the instruction at that position with a software interrupt...

Idea: Just replace the instruction at that position with a software interrupt...

```
6da: 55                push   %rbp
6db: 48 89 e5          mov    %rsp,%rbp
6de: 48 83 ec 30      sub    $0x30,%rsp
6e2: 89 7d dc          mov    %edi,-0x24(%rbp)
6e5: 48 89 75 d0      mov    %rsi,-0x30(%rbp)
6e9: 83 7d dc 01      cmpl  $0x1,-0x24(%rbp)
6ed: cc              int    $3
6ee: 0c              ???
6ef: 48 8d 3d 62 02 00 00 lea   0x262(%rip),%rdi
6f6: e8 95 fe ff ff   callq 590 <puts@plt>
6fb: c7 45 e8 00 00 00 00 movl  $0x0,-0x18(%rbp)
702: 48 c7 45 f0 00 00 00 movq  $0x0,-0x10(%rbp)
```

Idea: Just replace the instruction at that position with a software interrupt...

```
6da: 55                push   %rbp
6db: 48 89 e5          mov    %rsp,%rbp
6de: 48 83 ec 30      sub    $0x30,%rsp
6e2: 89 7d dc          mov    %edi,-0x24(%rbp)
6e5: 48 89 75 d0      mov    %rsi,-0x30(%rbp)
6e9: 83 7d dc 01      cmpl  $0x1,-0x24(%rbp)
6ed: cc              int    $3
6ee: 0c              ???
6ef: 48 8d 3d 62 02 00 00 lea   0x262(%rip),%rdi
6f6: e8 95 fe ff ff   callq 590 <puts@plt>
6fb: c7 45 e8 00 00 00 00 movl  $0x0,-0x18(%rbp)
702: 48 c7 45 f0 00 00 00 movq  $0x0,-0x10(%rbp)
```

Now we got the interrupt but how do we get the processor to execute our original instruction again?

Idea: The OS interrupt handler restores the original byte

Idea: The OS interrupt handler restores the original byte

```
6da: 55                push   %rbp
6db: 48 89 e5          mov    %rsp,%rbp
6de: 48 83 ec 30       sub    $0x30,%rsp
6e2: 89 7d dc          mov    %edi,-0x24(%rbp)
6e5: 48 89 75 d0       mov    %rsi,-0x30(%rbp)
6e9: 83 7d dc 01       cmpl  $0x1,-0x24(%rbp)
6ed: 7f                ???
6ee: 0c                ???
6ef: 48 8d 3d 62 02 00 00 lea   0x262(%rip),%rdi
6f6: e8 95 fe ff ff   callq 590 <puts@plt>
6fb: c7 45 e8 00 00 00 00 movl  $0x0,-0x18(%rbp)
702: 48 c7 45 f0 00 00 00 movq  $0x0,-0x10(%rbp)
```

Idea: The OS interrupt handler restores the original byte

```
6da: 55                push   %rbp
6db: 48 89 e5          mov    %rsp,%rbp
6de: 48 83 ec 30       sub    $0x30,%rsp
6e2: 89 7d dc          mov    %edi,-0x24(%rbp)
6e5: 48 89 75 d0       mov    %rsi,-0x30(%rbp)
6e9: 83 7d dc 01       cmpl  $0x1,-0x24(%rbp)
6ed: 7f                ???
6ee: 0c                ???
6ef: 48 8d 3d 62 02 00 00 lea   0x262(%rip),%rdi
6f6: e8 95 fe ff ff   callq 590 <puts@plt>
6fb: c7 45 e8 00 00 00 00 movl  $0x0,-0x18(%rbp)
702: 48 c7 45 f0 00 00 00 movq  $0x0,-0x10(%rbp)
```

But we are at the wrong location... What now?

Idea: The OS can manipulate the `%eip/%rip`

Idea: The OS can manipulate the `%eip/%rip`

```
6da: 55          push   %rbp
6db: 48 89 e5    mov    %rsp,%rbp
6de: 48 83 ec 30 sub    $0x30,%rsp
6e2: 89 7d dc    mov    %edi,-0x24(%rbp)
6e5: 48 89 75 d0 mov    %rsi,-0x30(%rbp)
6e9: 83 7d dc 01 cmpl   $0x1,-0x24(%rbp)
6ed: 7f 0c      jg     6fb <main+0x21>
6ef: 48 8d 3d 62 02 00 00 lea   0x262(%rip),%rdi
6f6: e8 95 fe ff ff      callq 590 <puts@plt>
6fb: c7 45 e8 00 00 00 00 movl  $0x0,-0x18(%rbp)
702: 48 c7 45 f0 00 00 00 movq  $0x0,-0x10(%rbp)
```

```
int main(int argc, char* argv[]) {
    if (argc <= 1)
        printf("usage: ./tohex <number> <base>\n");
    unsigned int base = 0;
    for (ssize_t i = 0; i < strlen(argv[2]); ++i) {
        base *= 10;
        base = argv[2][i] - '0';
    }
    printf("base = %u\n",base);
    unsigned int result = 0;
    for (ssize_t i = 0; i < strlen(argv[1]); ++i) {
        asm("int $3");
        result *= 10;
        if (argv[1][i] >= '0' && argv[1][i] <= '9')
            result = argv[1][i] - '0';
        if (argv[1][i] >= 'A' && argv[1][i] <= 'Z')
            result = argv[1][i] - 'A';
        if (argv[1][i] >= 'a' && argv[1][i] <= 'z')
            result = argv[1][i] - 'a';
    }
}
```

```
daniel@lava: cgdb ./a.out
File Edit View Search Terminal Help
16 unsigned int result = 0;
17 for (ssize_t i = 0; i < strlen(argv[1]); ++i)
18 {
19     asm("int $3");
20 -> result *= 10;
21     if (argv[1][i] >= '0' && argv[1][i] <= '9')
22         result = argv[1][i] - '0';
23     if (argv[1][i] >= 'A' && argv[1][i] <= 'Z')
/home/daniel/example.c
Reading symbols from ./a.out...done.
(gdb) run FFF 16
Starting program: /home/daniel/a.out FFF 16
base = 6

Program received signal SIGTRAP, Trace/breakpoint trap.
main (argc=3, argv=0x7fffffffde68) at example.c:20
(gdb)
```

- One of the most powerful applications of version control
- Bisection search (binary search) over commits to find a specific change (such as a bug)
- Efficient: $O(n) = \log(n)$

- 2005 commits in SWEB
- Student reports a reproducible bug
- Unclear what the source of the bug is

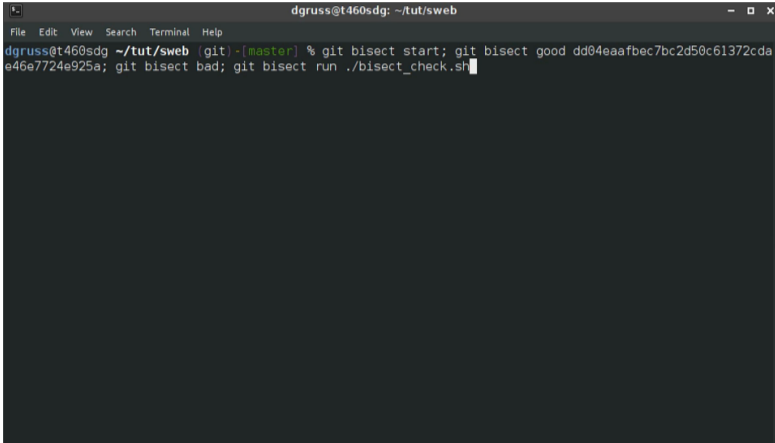

```
daniel@lava /d/arbeit/sweb (git)-[master] % git bisect start
daniel@lava /d/arbeit/sweb (git)-[master|bisect] % git bisect bad
daniel@lava /d/arbeit/sweb (git)-[master|bisect] % git log --oneline | tail -n 1
dd04eaaf Initial revision
daniel@lava /d/arbeit/sweb (git)-[master|bisect] % git bisect good dd04eaaf
Bisecting: 1001 revisions left to test after this (roughly 10 steps)
[237a3530abd066f02322ae76010d4bec5330eec7] unlocked load_lock_
daniel@lava /d/arbeit/sweb (git)-[237a353...|bisect] %
```

- 2005 commits → only ≈ 10 revisions to test
- still too lazy to manually test 10 revisions?

```
#!/bin/bash
(rm -rf ../sweb-bin && cmake . && make -j4) || (git clean -f -d -X && rm -rf /
    tmp/sweb && mkdir -p /tmp/sweb && cd /tmp/sweb && cmake ~/tut/sweb && make
    -j4)
if [ $? -eq 0 ]; then
    exit 1
fi
exit 0
```

- git bisect run:

```
#!/bin/bash
git bisect start
git bisect good dd04eaafb7bc2d50c61372cdae46e7724e925a
git bisect bad
git bisect run ./bisect_check.sh
```

A terminal window with a dark background and light text. The title bar reads "dgruss@t460sdg: ~/tut/sweb". The menu bar includes "File", "Edit", "View", "Search", "Terminal", and "Help". The main content shows a command sequence: "dgruss@t460sdg ~/tut/sweb (git)-[master] % git bisect start; git bisect good dd04eaafbec7bc2d50c61372cda e46e7724e925a; git bisect bad; git bisect run ./bisect_check.sh". The cursor is at the end of the last command.

```
dgruss@t460sdg: ~/tut/sweb (git)-[master] % git bisect start; git bisect good dd04eaafbec7bc2d50c61372cda e46e7724e925a; git bisect bad; git bisect run ./bisect_check.sh
```

Proper debugging may be impossible for various reasons

- Booting a kernel
- Race conditions disappear if timing changes

- printf in interesting lines

- printf in interesting lines
- many lines → see how far it gets

- printf in interesting lines
- many lines → see how far it gets
- more extreme form: print register debugging

- printf in interesting lines
- many lines → see how far it gets
- more extreme form: print register debugging
 - if you can only get register values upon crash and nothing else

- printf in interesting lines
- many lines → see how far it gets
- more extreme form: print register debugging
 - if you can only get register values upon crash and nothing else
- most extreme form: LED debugging

- printf in interesting lines
- many lines → see how far it gets
- more extreme form: print register debugging
 - if you can only get register values upon crash and nothing else
- most extreme form: LED debugging
 - if you can only get LED on or off as feedback

Remember when you explained a problem to someone and while explaining it, you found the solution yourself?



- Save your colleague some time

- Save your colleague some time
- Explain it to the duck as if it was a colleague

- Save your colleague some time
- Explain it to the duck as if it was a colleague
- Speaking aloud helps



- Debugging needs experts



- Debugging needs experts
- Experts are expensive






- Debugging needs experts
- Experts are expensive
- Debugging is expensive

Questions?



99 little bugs in the code.
99 little bugs in the code.
Take one down, patch it around.
127 little bugs in the code...

-  Cormac Flanagan and Stephen N Freund.
Fasttrack: efficient and precise dynamic race detection.
In ACM Sigplan Notices, volume 44, pages 121–133. ACM, 2009.
-  Timur Iskhodzhanov, Alexander Potapenko, Alexey Samsonov, Kostya Serebryany, Evgeniy Stepanov, and Dmitriy Vyukov.
Finding races and memory errors with llvm instrumentation.
2011.
-  OWASP.
Application Threat Modeling.