C/C++ VS Security!

Gynvael Coldwind
code::dive, Wrocław, 2018

If you are reading these slides (i.e. not watching the actual talk)
please note there are speaker notes under each slide.

About your presenter

Tech Lead / Manager @ Google ISE-RIP

All opinions expressed during this presentation are mine and mine alone, and not those of my barber, my accountant or my employer.

What's on the menu

Also featuring:
Other languages
A lot of sidenotes

The devil is in the details.

- Pitfalls
- Traps
- Perils
- Hazards
- Errors
- And in general, bugs.

Not to be confused with "C/C++ vs Security" talk by Michał "Redford" Kowalczyk - check it out though!

(note the details in Speaker Notes section of each slide)

Full disclosure: This talk contains a couple of (updated) slides from my other talks.

Special thanks to:

- **KrzaQ** (https://krzaq.cc)
- **disconnect3d** (https://disconnect3d.pl/)
- **Redford**
- **foxtrot_charlie** (http://foxtrotlabs.cc)
- **j00ru** (https://j00ru.vexillium.org)
- and probably A LOT of other people :)

Old-style C/C++ again?!

But the previous programmer's code isn't as good as mine!

Modern C++ is safer.

- C++ Core Guidelines

But there is a lot of older code we have to deal with.

- Can't write everything from scratch.
- Can't rewrite everything from scratch.

But it was NOT INVENTED HERE!

Uhm, actually, why is C/C++ code so bug-ridden?

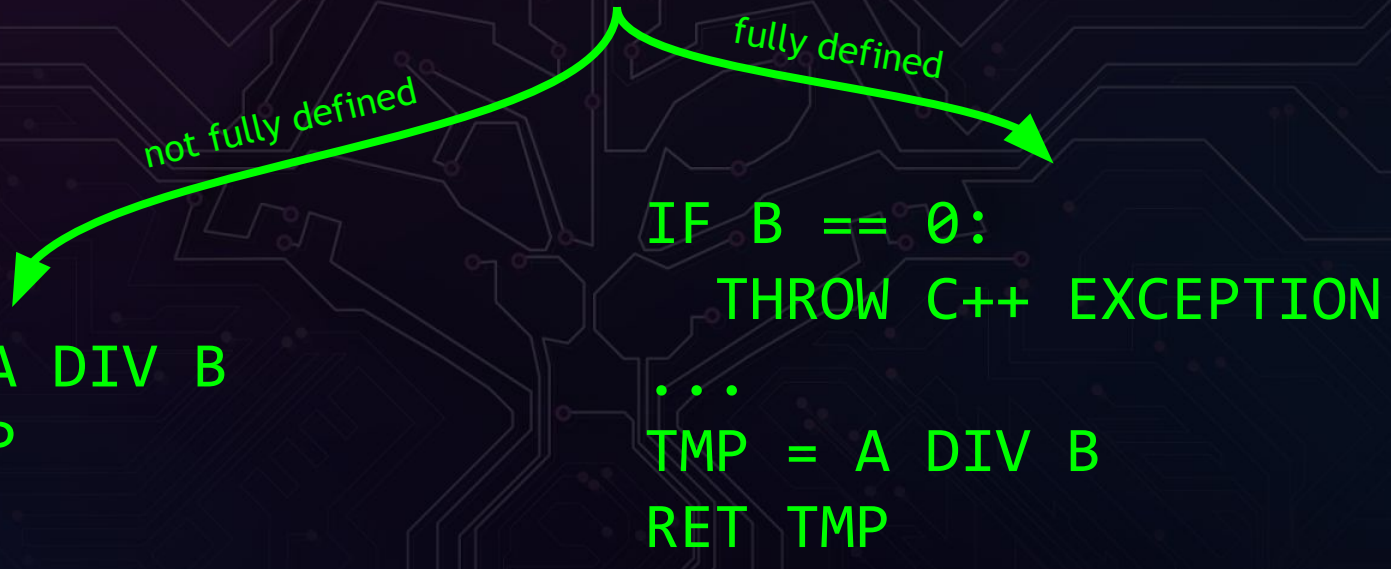
It's because it's **fast** and **general purposed**!

- Not fully defined in the standard (UB, UNSB, IB)
- "as if" rule
- Generalized execution/memory model

Glossary: UB (Undefined Behavior)

The compiler assumes your code doesn't do anything weird.

```
int div(int a, int b) { return a / b; }
```



BTW: How to secure this code?

Fixed!

```
std::optional<int> my_div(int a, int b) {
  if (b == 0) {
    return {};
  }

  return a / b;
}
```

BTW: Int ranges.

Two's Complement signed integers have one more negative value (example for 32-bit ints).

2147483647

-2147483648

BTW: Int ranges and literal types.

Two's Complement signed integers have one more negative value (example for 32-bit ints).

2147483647 int

-(2147483648) long

-2147483647 - 1 int

Glossary: UNSB (Unspecified Behavior)

The standard gives a range of things that may happen, but leaves it to the implementation to select the specific effect.

```
int res = a() + b() + c();
```

```
exec a() →Ta
exec b() →Tb
eval Ta + Tb → Tab
exec c() → Tc
eval Tab + Tc → res
```

```
exec c() → Tc
exec b() →Tb
exec a() →Ta
eval Tb + Tc → Tbc
eval Ta + Tbc → res
```

```
int res = a() || b() || c();
```

Glossary: IB (Implementation-defined Behavior)

The compiler decides what happens (and it can be anything).

- fopen and "r" vs "rb"
  - most *nix: no difference
  - Windows: CR/LF, ^Z (1A)
- sizes of various variables
  - LLP64: long is 32-bit
  - LP64: long is 64-bit

BTW: UB → UNSB → IB

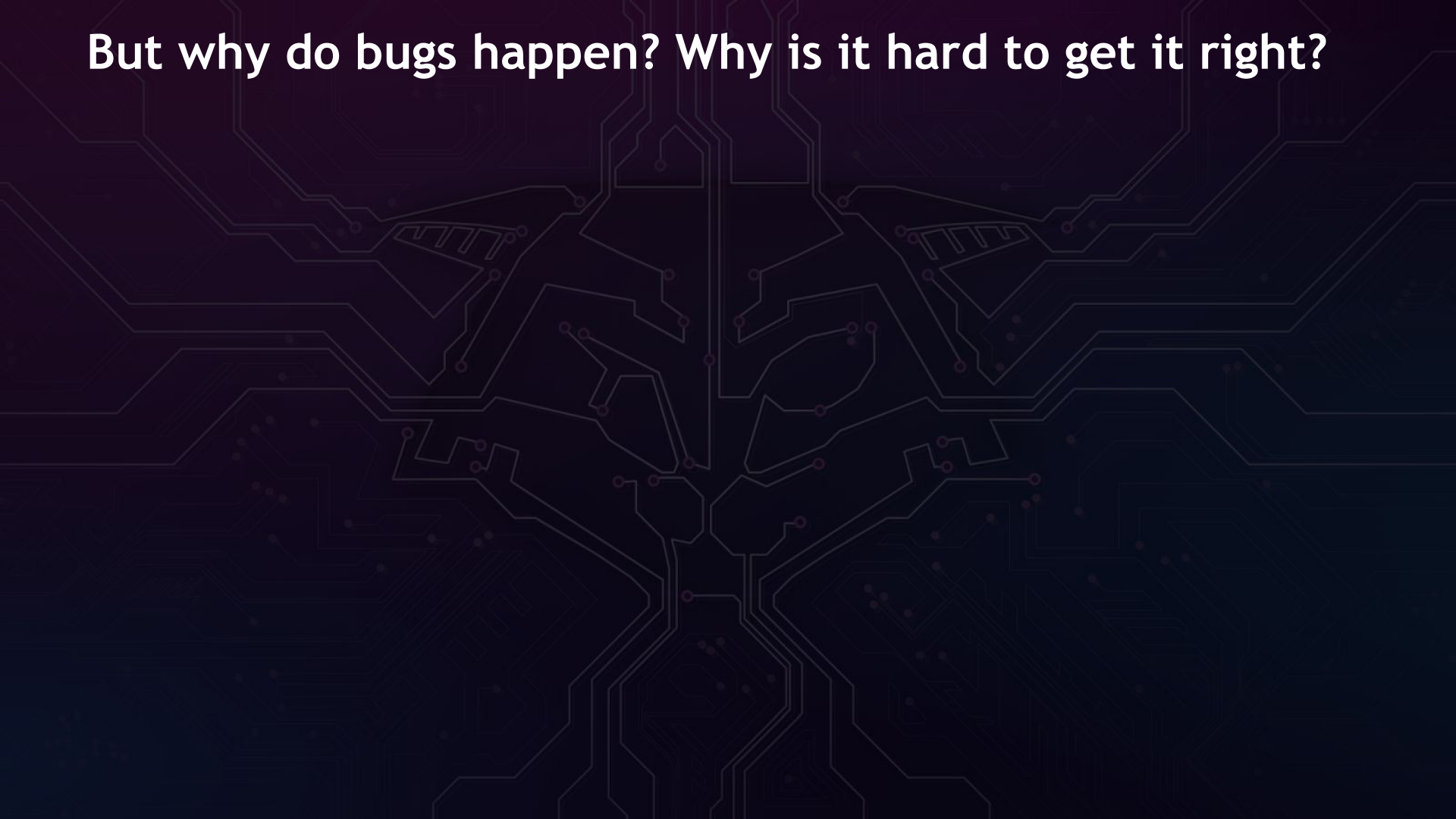
To sum up UB/INSB/IB...

Relying on UNSB/IB → usually portability issues.

Relying on UB → asking for trouble / getting hacked.

"Programmers shouldn't discuss what the compiler does on an UB and should avoid UBs in the code."

But why do bugs happen? Why is it hard to get it right?



Discrepancy in execution

1. What the programmer meant.

Discrepancy in execution

1. What the programmer meant.
2. What the source code says will happen.

Discrepancy in execution

1. What the programmer meant.
2. What the source code says will happen.
3. What the standard says will happen.

Discrepancy in execution

1. What the programmer meant.
2. What the source code says will happen.
3. What the standard says will happen.
4. What the generated assembly looks like.

Discrepancy in execution

1. What the programmer meant.
2. What the source code says will happen.
3. What the standard says will happen.
4. What the generated assembly looks like.
5. What the actual (dis)assembly looks like.

Discrepancy in execution

1. What the programmer meant.
2. What the source code says will happen.
3. What the standard says will happen.
4. What the generated assembly looks like.
5. What the actual (dis)assembly looks like.
6. What the CPU actually executes.

Discrepancy in execution

1. What the programmer meant.
2. What the source code says will happen.

Discrepancy in execution

VS

1. What the programmer meant.
2. What the source code says will happen.

Returns 0 when both blocks are equal

```
int8_t result = memcmp(pwd_hash, SECRET, 32);

if (result == 0) {
  // GOOD PASSWORD!
}
```

https://nakedsecurity.sophos.com/2012/06/13/anatomy-of-a-bug-the-mysql-authentication-disaster-patch-now/

Discrepancy in execution

VS

1. What the programmer meant.
2. What the source code says will happen.

A good programmer knows this!

Most implementations return difference between two unequal bytes.

```
int8_t result = memcmp(pwd_hash, SECRET, 32);

if (result == 0) {
  // GOOD PASSWORD!
}
```

https://nakedsecurity.sophos.com/2012/06/13/anatomy-of-a-bug-the-mysql-authentication-disaster-patch-now/

Discrepancy in execution

VS

1. What the programmer meant.
2. What the source code says will happen.

A great programmer knows this!

Actually returns an int.
Specified: != 0 or == 0.

```
int8_t result = memcmp(pwd_hash, SECRET, 32);

if (result == 0) {
  // GOOD PASSWORD!
}
```

https://nakedsecurity.sophos.com/2012/06/13/anatomy-of-a-bug-the-mysql-authentication-disaster-patch-now/

Discrepancy in execution

VS

1. What the programmer meant.
2. What the source code says will happen.

Actually returns an int.
Specified: != 0 or == 0.

```
int8_t result = memcmp(pwd_hash, SECRET, 32);

if (result == 0) {
  // GOOD PASSWORD!
}
```

So it can return e.g. 512

https://nakedsecurity.sophos.com/2012/06/13/anatomy-of-a-bug-the-mysql-authentication-disaster-patch-now/

Discrepancy in execution

VS

1. What the programmer meant.
2. What the source code says will happen.

Actually returns an int.
Specified: != 0 or == 0.

```
int8_t result = memcmp(pwd_hash, SECRET, 32);

if (result == 0) {
  // GOOD PASSWORD!
}
```

So it can return e.g. 512

0x200

https://nakedsecurity.sophos.com/2012/06/13/anatomy-of-a-bug-the-mysql-authentication-disaster-patch-now/

Discrepancy in execution

VS

1. What the programmer meant.
2. What the source code says will happen.

Actually returns an int.
Specified: != 0 or == 0.

```
int8_t result = memcmp(pwd_hash, SECRET, 32);

if (result == 0) {
  // GOOD PASSWORD!
}
```

So it can return e.g. 512

int8_t(0x200) is 0x00 → 0x200

https://nakedsecurity.sophos.com/2012/06/13/anatomy-of-a-bug-the-mysql-authentication-disaster-patch-now/

Discrepancy in execution

VS

1. What the programmer meant.
2. What the source code says will happen.

CVE-2012-2122
(optimized memcmp)

```
$ for i in `seq 1 1000`
> do
> mysql -u root --password=$i -h HOST 2>/dev/null
> done
mysql>
```

Technically: Integer Truncation (UB) → Authentication Bypass

https://community.rapid7.com/community/metasploit/blog/2012/06/11/cve-2012-2122-a-tragically-comedic-security-flaw-in-mysql

Discrepancy in execution

VS

2. What the source code says will happen.
3. What the standard says will happen.

```
int add_no_overflow(int a, int b) {
    if (a < 0) return -1;
    if (b < 0) return -1;
    // Make sure no overflow happened!
    if (a + b < 0) return -1;
    return a + b;
}
```

a and b must be positive or zero

in case of overflow the value will be negative!

Discrepancy in execution

VS

2. What the source code says will happen.
3. What the standard says will happen.

```
int add_no_overflow(int a, int b) {
    if (a < 0) return -1;
    if (b < 0) return -1;
    // Make sure no overflow happened!
    if (a + b < 0) return -1;
    return a + b;
}
```

DEMO

Discrepancy in execution

2. What the source code says will happen.
3. What the standard says will happen.


```
printf("testing %i + %i --> %i\n",
    std::numeric_limits<int>::max(), 42,
    add_no_overflow(
       std::numeric_limits<int>::max(), 42));
```

Optimized to:

```
printf("testing %i + %i → %i\n",
    2147483647, 41, -1);
```

Discrepancy in execution

VS

2. What the source code says will happen.
3. What the standard says will happen.

What just happened???

```
int add_no_overflow(int a, int b) {
    if (a < 0) return -1;
    if (b < 0) return -1;
    // Make sure no overflow happened!
    if (a + b < 0) return -1;
    return a + b;
}
```

a and b must be positive or zero
(and now the compiler knows that too)

two non-negative numbers will always give a non-negative number
this is deadcode! (but overflow...?)

Discrepancy in execution

2. What the source code says will happen.
3. What the standard says will happen.

```
int32_t add_no_overflow(int32_t a, int32_t b) {
    if (a < 0) return -1;
    if (b < 0) return -1;
    int64_t res = (int64_t)a + (int64_t)b;
    if (res > INT_MAX) return -1;
    return (int32_t)res;
}  // Or just use a safe int library.
```

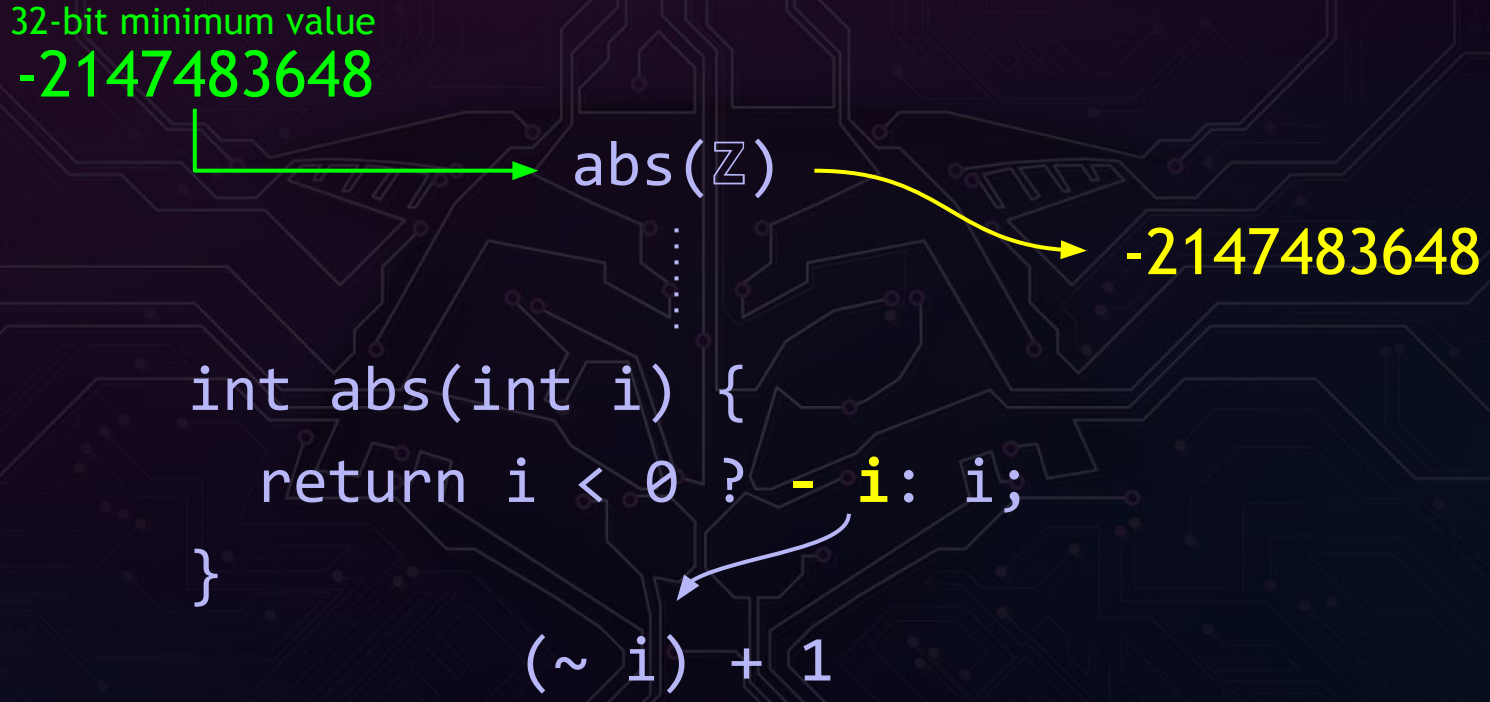
Speaking of integer overflows...


```
int abs(int i) {
  return i < 0 ? - i: i;
}
```

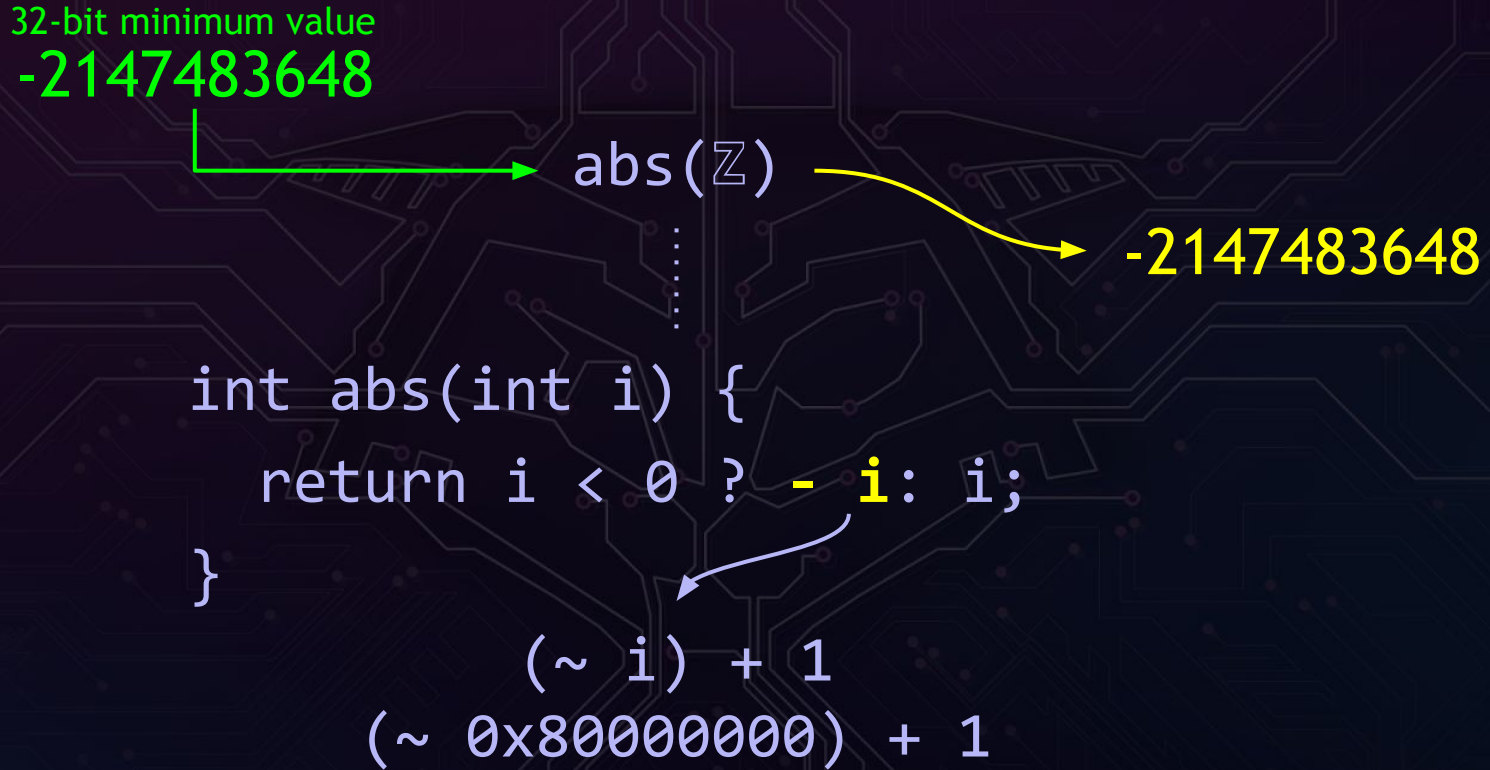
Speaking of integer overflows...


```
int abs(int i) {
  return i < 0 ? - i: i;
}
```

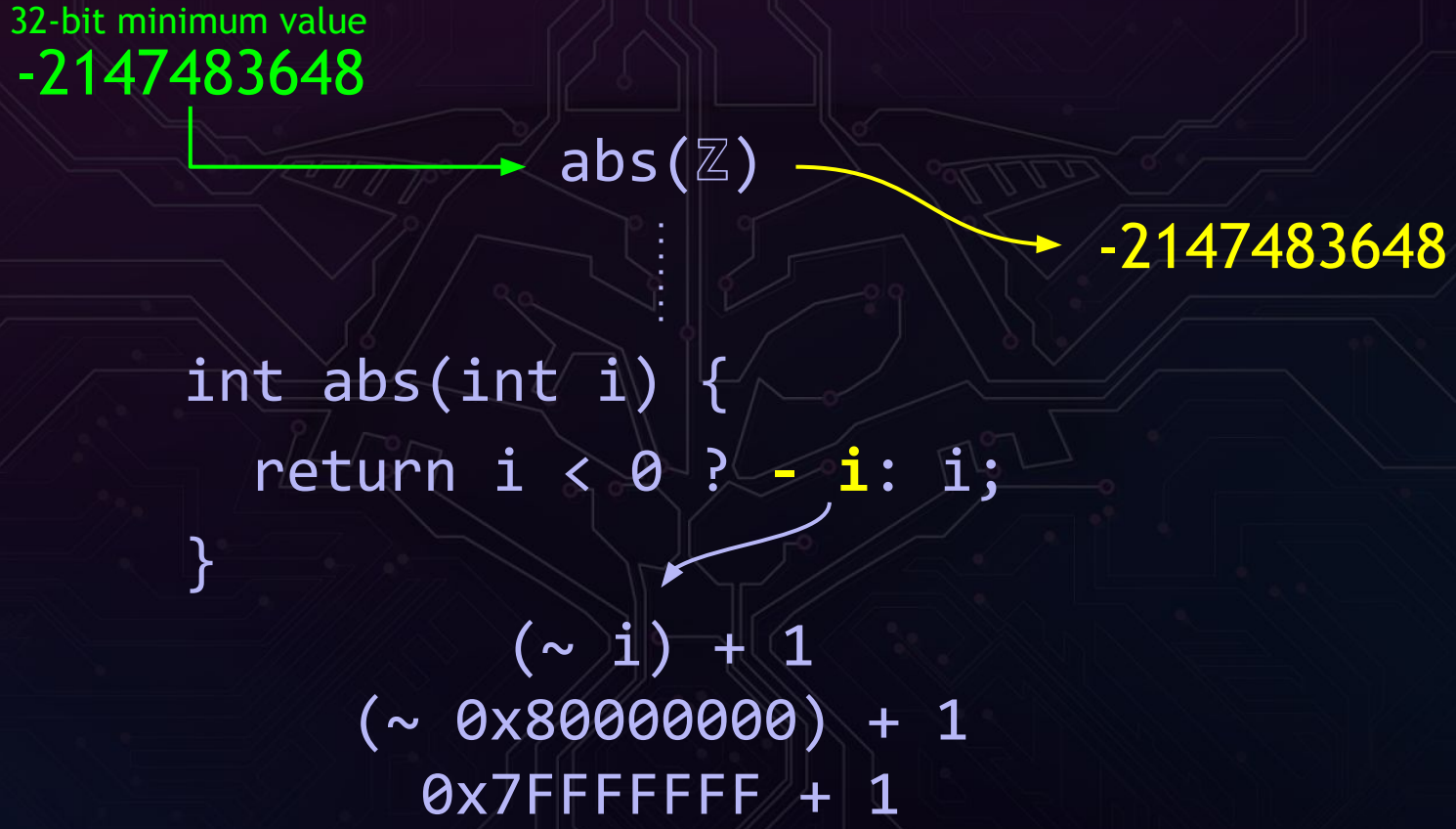
Speaking of integer overflows...



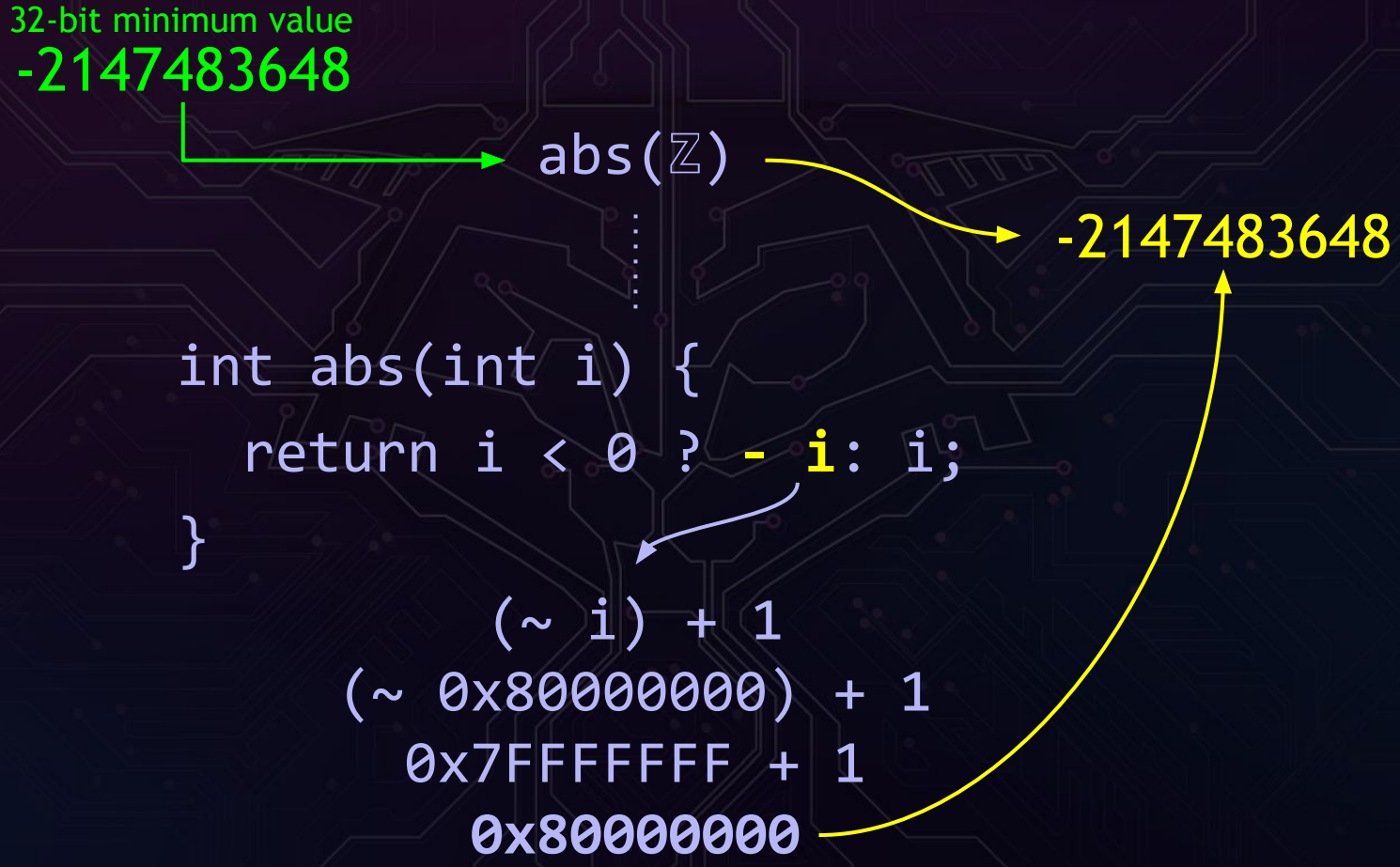
Speaking of integer overflows...



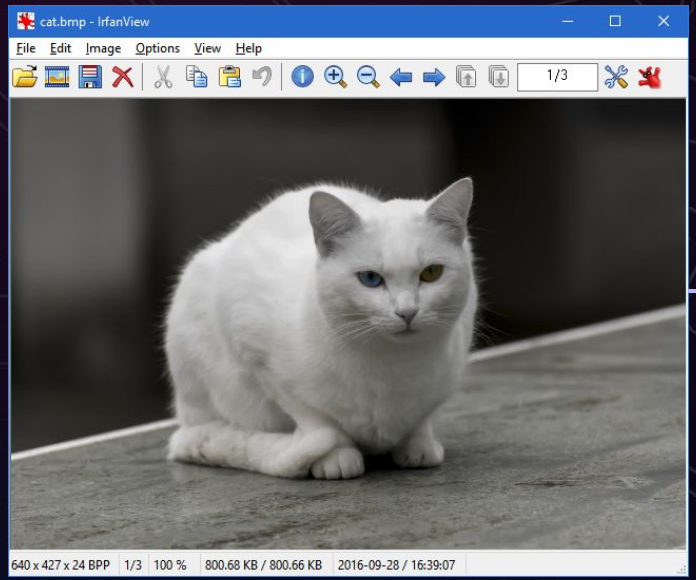
Speaking of integer overflows...

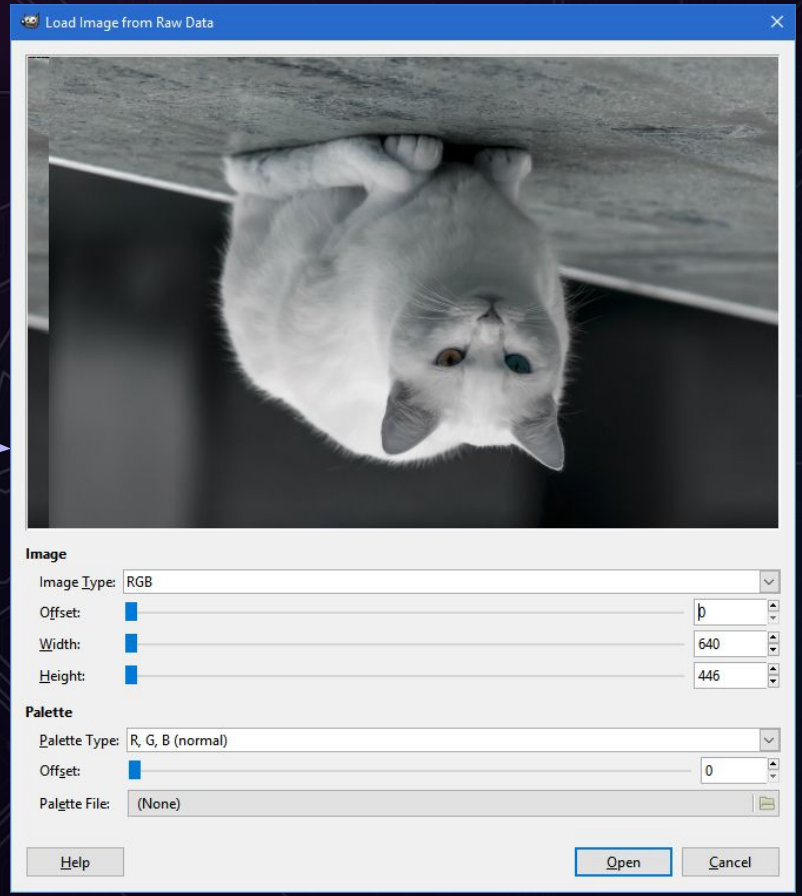


Speaking of integer overflows...

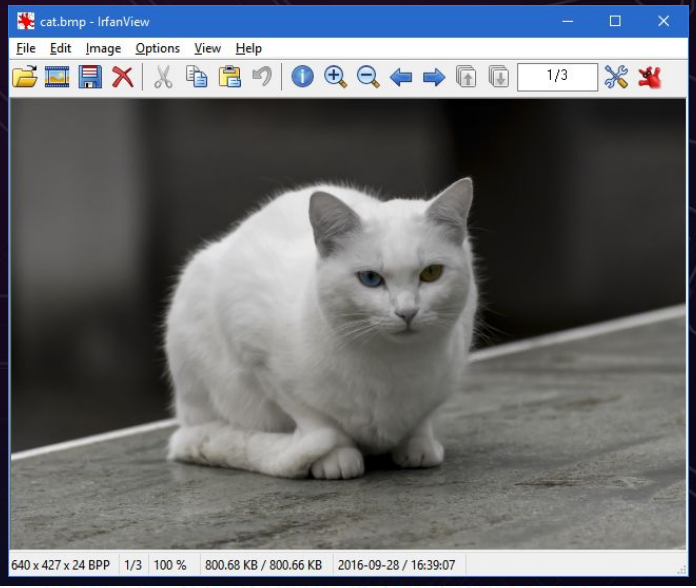


Speaking of integer overflows... abs() and BMP





Speaking of integer overflows... abs() and BMP



427 "upside down"

-427 "normal"

0000000E struct tagBITMAPINFOHEADER	{...}
0000000E uint32 biSize	40
00000012 int32 biWidth	640
00000016 int32 biHeight	427

Speaking of integer overflows... abs() and BMP

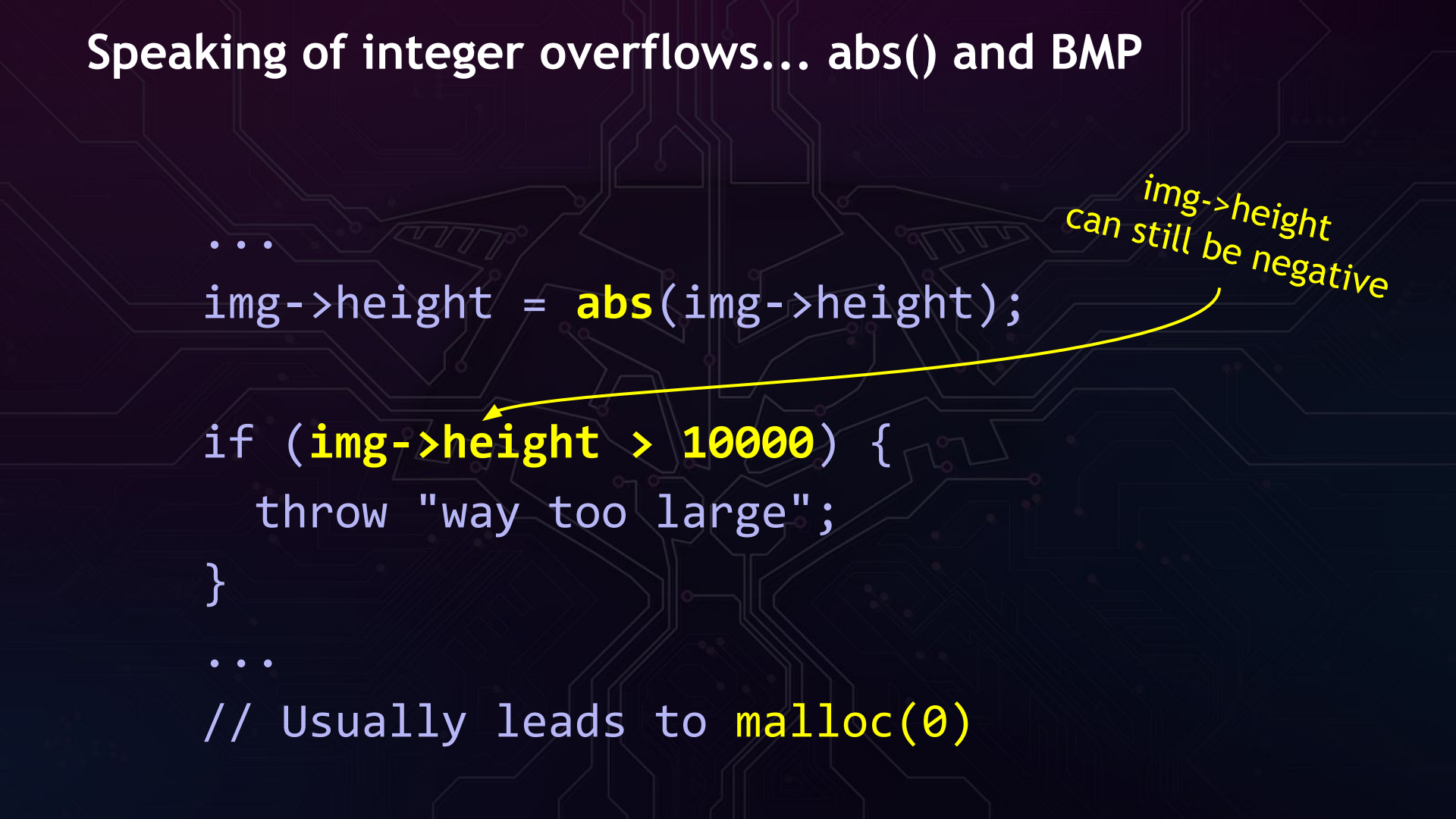
```
bool upside_down = true;
if (img->height < 0) {
  upside_down = false;
}

img->height = abs(img->height);
...
```

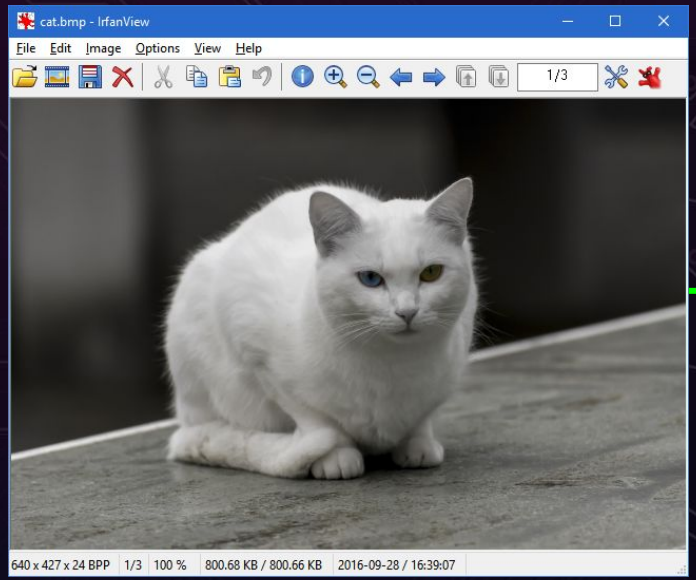
Speaking of integer overflows... abs() and BMP

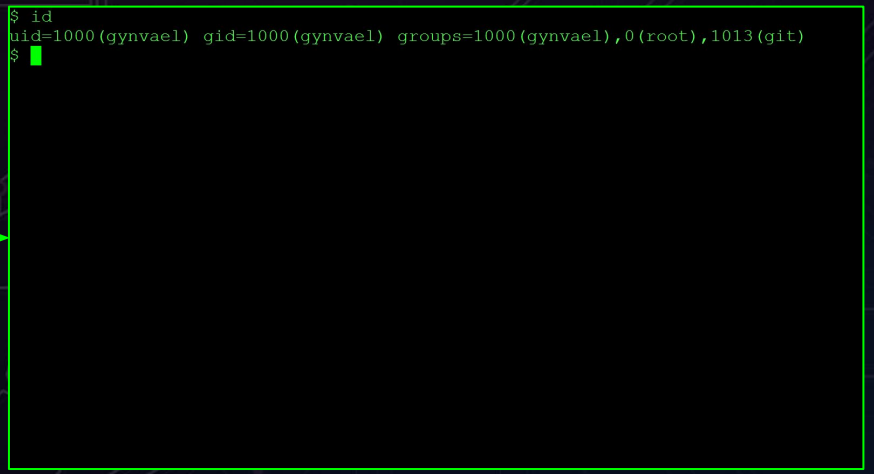
```
...
img->height = abs(img->height);

if (img->height > 10000) {
  throw "way too large";
}
...
// Usually leads to malloc(0)
```



Speaking of integer overflows... abs() and BMP





We're watching an image.
Somebody is watching us ;o

Discrepancy in execution

2. What the source code says will happen.
4. What the generated assembly looks like.

Discrepancy in execution

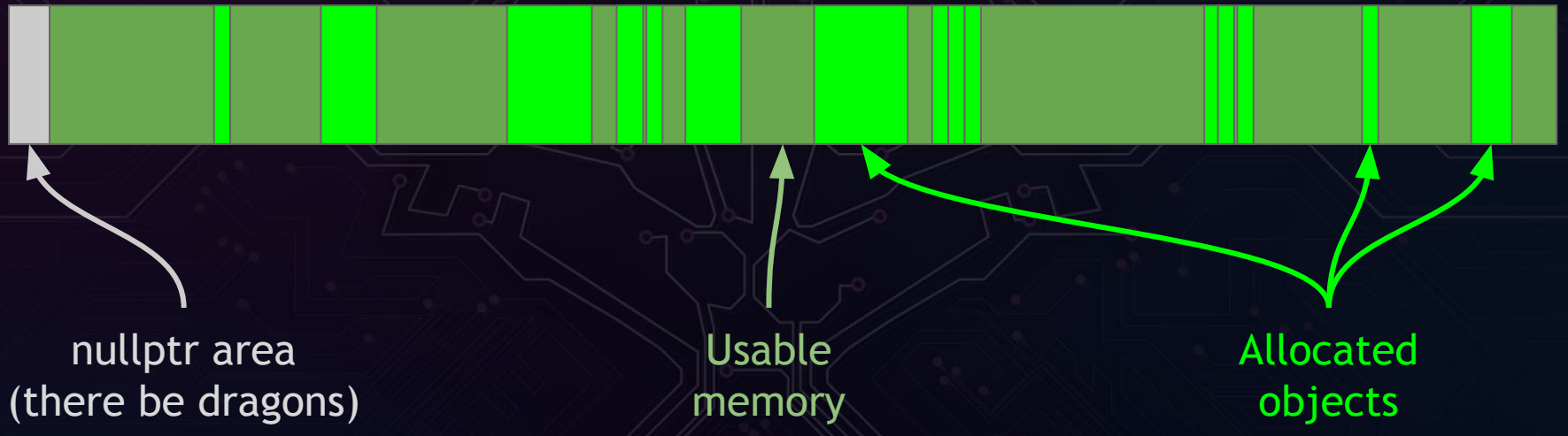
VS

2. What the source code says will happen.
4. What the generated assembly looks like.

but first, a primer on double-fetches!

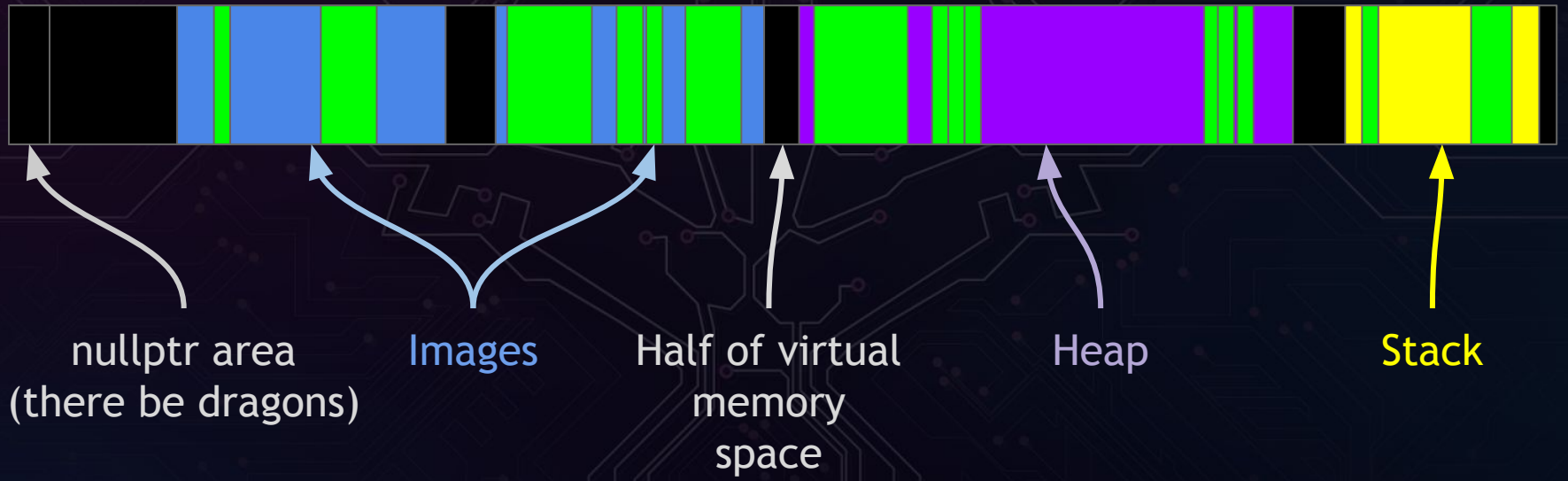
A primer on double-fetches

Standard C/C++ memory model:



A primer on double-fetches

A typical (though simplified) C/C++ memory model:



A primer on double-fetches

C/C++ and shared memory between privileges:

E.g. kernel & user apps, hypervisor and VMs, two processes sharing memory.

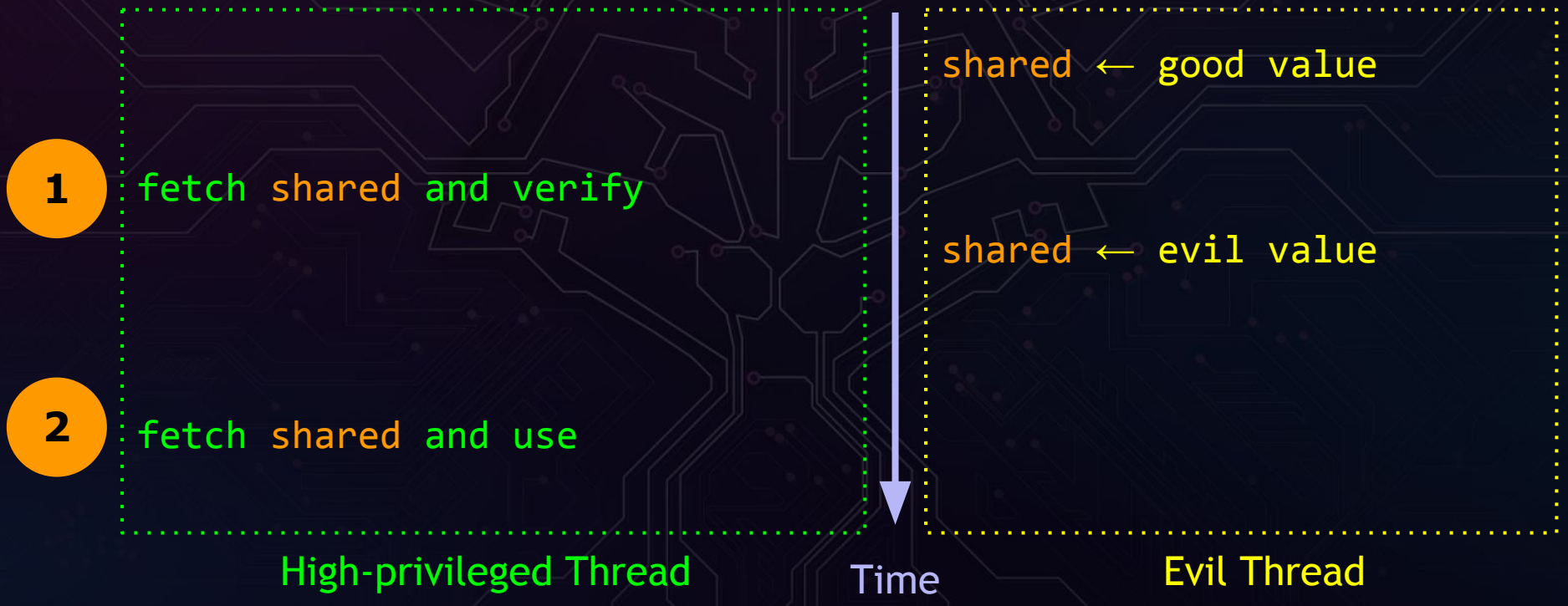
Memory owned and used by the less privileged code (thread)

Shared memory used to communicate (usually owned by less privileged thread)

A primer on double-fetches

A classical "TOCTTOU" double-fetch vulnerability:

leads to: buffer overflow of various flavors (e.g. when shared is a buffer size)



A primer on double-fetches

"Wait wait! That's just like 3 CPU cycles time window!"

3-4 CPU cycles ?

greatly simplified!

```
mov   eax, DWORD PTR [ebx]
add   eax, -128
jle   L14
L14:
mov   ecx, DWORD PTR [ebx]
```

A primer on double-fetches

"Wait wait! That's just like 3 CPU cycles time window!"

7-8 CPU cycles

greatly simplified!

```
mov   eax, DWORD PTR [ebx]
add   eax, -128
jle   L14
L14:
mov   ecx, DWORD PTR [ebx]
```

Cache line boundary

A primer on double-fetches

"Wait wait! That's just like 3 CPU cycles time window!"

14 CPU cycles

greatly simplified!

```
mov   eax, DWORD PTR [ebx]
add   eax, -128
jle   L14
L14:
mov   ecx, DWORD PTR [ebx]
```

Cache line boundary + L1 cache miss

A primer on double-fetches

"Wait wait! That's just like 3 CPU cycles time window!"

50 CPU cycles

greatly simplified!

```
mov   eax, DWORD PTR [ebx]
add   eax, -128
jle   L14
L14:
mov   ecx, DWORD PTR [ebx]
```

Cache line boundary + L1+L2 cache miss

A primer on double-fetches

"Wait wait! That's just like 3 CPU cycles time window!"

200 CPU cycles

greatly simplified!

```
mov   eax, DWORD PTR [ebx]
add   eax, -128
jle   L14
L14:
mov   ecx, DWORD PTR [ebx]
```

Cache line boundary + L1+L2+L3 cache miss

A primer on double-fetches

"Wait wait! That's just like 3 CPU cycles time window!"

400 CPU cycles

greatly simplified!

```
mov   eax, DWORD PTR [ebx]
add   eax, -128
jle   L14
L14:
mov   ecx, DWORD PTR [ebx]
```

RAM block / Page boundary + cache disabled / L3 cache miss

A primer on double-fetches

"Wait wait! That's just like 3 CPU cycles time window!"

2000 CPU cycles

greatly simplified!

```
mov   eax, DWORD PTR [ebx]
add   eax, -128
jle   L14
L14:
mov   ecx, DWORD PTR [ebx]
```

Page boundary + cache disabled / L3 cache miss + TLB miss

A primer on double-fetches

"Wait wait! That's just like 3 CPU cycles time window!"

milliseconds

```
mov   eax, DWORD PTR [ebx]
add   eax, -128
jle   L14
L14:
mov   ecx, DWORD PTR [ebx]
```

Page swapped out

A primer on double-fetches

Yeah, should be enough time ;)

milliseconds

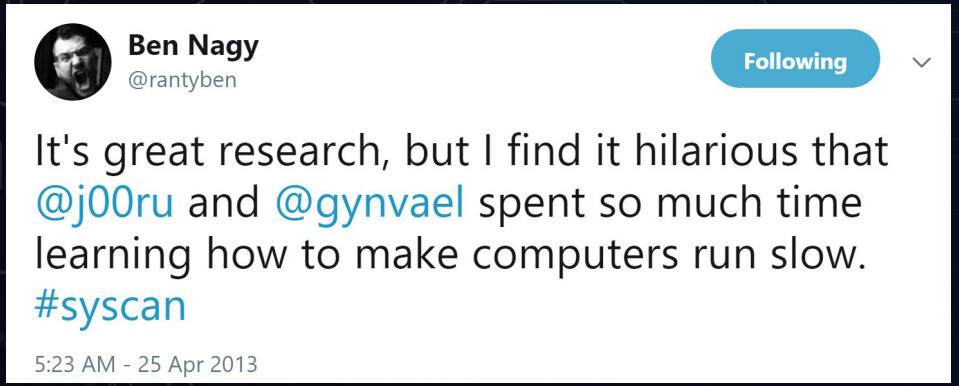
```
mov   eax, DWORD PTR [ebx]
add   eax, -128
jle   L14
L14:
mov   ecx, DWORD PTR [ebx]
```

Page swapped out

BTW: Bochspwn

did you go to the FPGA talk today?

Can be detected with an instrumented emulator at runtime, e.g. **bochspwn** research project by j00ru & me and A LOT of follow ups by various researchers (e.g. xenpwn).

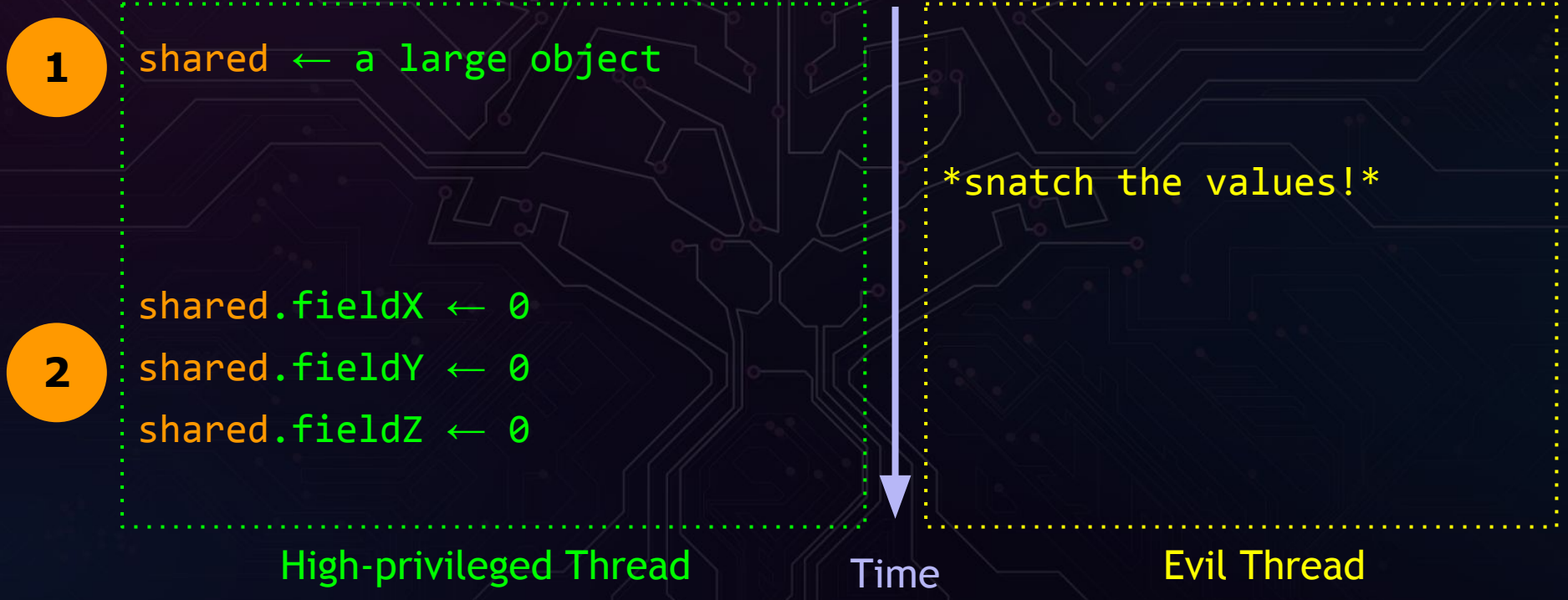


https://github.com/googleprojectzero/bochspwn

A primer on double-fetches

A double-write info-leak (yes, these happen in the wild):

Oh btw, https://j00ru.vexillium.org/papers/2018/bochspwn_reloaded.pdf



Discrepancy in execution

VS

2. What the source code says will happen.
4. What the generated assembly looks like.

back to our show

https://www.ernw.de/download/xenpwn.pdf (Felix Wilhelm)

Discrepancy in execution

VS

2. What the source code says will happen.
4. What the generated assembly looks like.

```
switch (u->type) {
  case TYPE_A:
    // Code.
    break;

  case TYPE_B:
    // Code.
    break;
}
```

Looking at the source code:
Only one fetch.

https://www.ernw.de/download/xenpwn.pdf (Felix Wilhelm)

Discrepancy in execution

VS

2. What the source code says will happen.
4. What the generated assembly looks like.

```
switch (u->type) {
  case TYPE_A:
    // Code.
    break;

  case TYPE_B:
    // Code.
    break;
}
```

Reasonable thing to expect:
1. Fetch u->type into a CPU register.
2. Compare vs TYPE_A, jump if equal.
3. Compare vs TYPE_B, jump if equal.

https://www.ernw.de/download/xenpwn.pdf (Felix Wilhelm)

Discrepancy in execution

VS

2. What the source code says will happen.
4. What the generated assembly looks like.

```
switch (u->type) {
  case TYPE_A:
    // Code.
    break;

  case TYPE_B:
    // Code.
    break;
}
```

Reasonable thing to expect:
1. Fetch u->type into a CPU register.
2. Compare against range,
   jump if outside.
3. Use a jump table[reg].

https://www.ernw.de/download/xenpwn.pdf (Felix Wilhelm)

Discrepancy in execution

VS

2. What the source code says will happen.
4. What the generated assembly looks like.

```
switch (u->type) {
  case TYPE_A:
    // Code.
    break;

  case TYPE_B:
    // Code.
    break;
}
```

What actually happens is unspecified ofc.

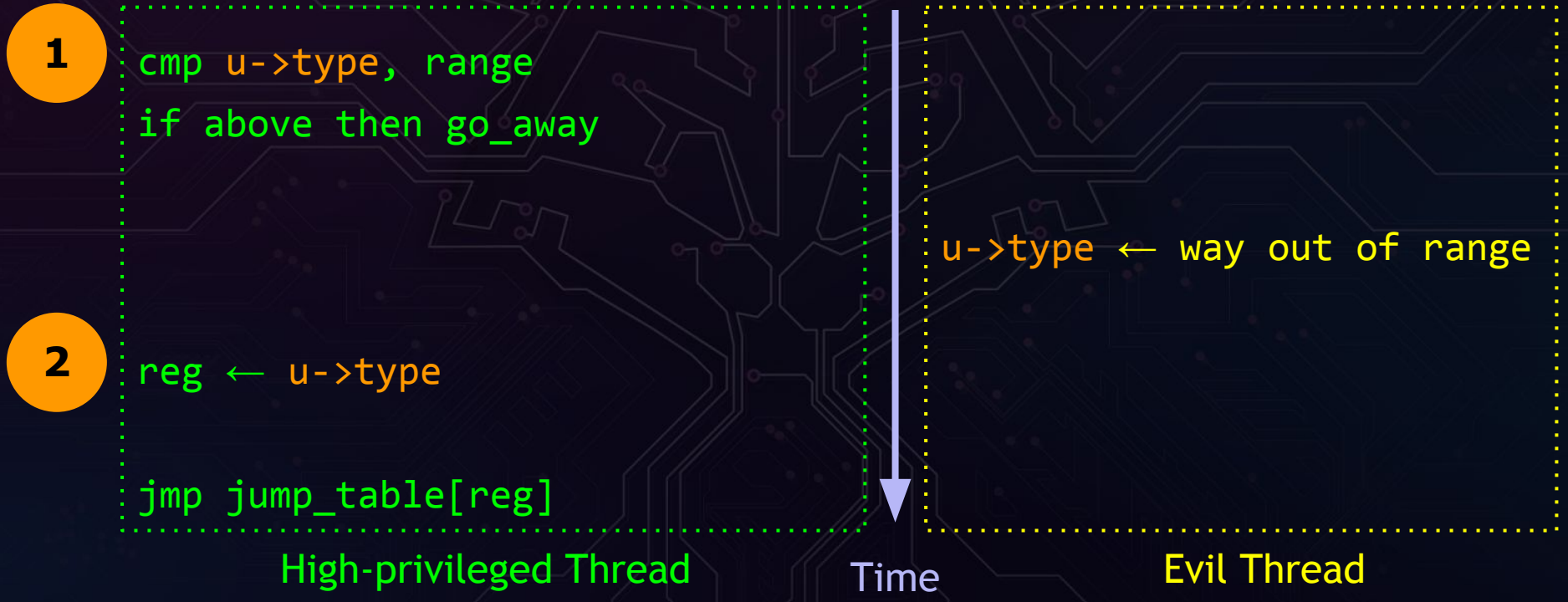
1. Compare u->type vs range,
   jump if outside.
2. Fetch u->type into a CPU register.
3. Use a jump table[reg].

https://www.ernw.de/download/xenpwn.pdf (Felix Wilhelm)

Discrepancy in execution

VS

2. What the source code says will happen.
4. What the generated assembly looks like.



Discrepancy in execution

VS

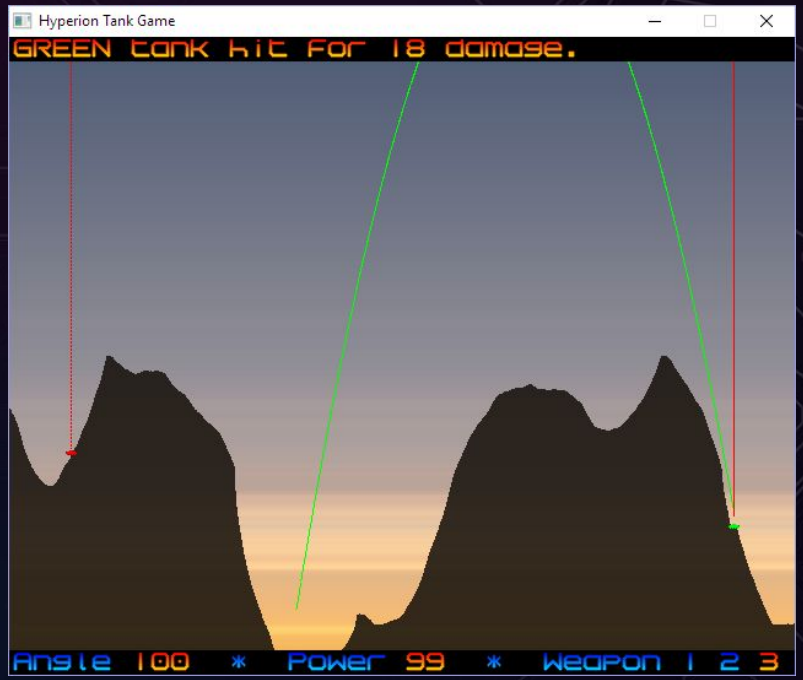
2. What the source code says will happen.
4. What the generated assembly looks like.

Another example.

Discrepancy in execution

VS

2. What the source code says will happen.
4. What the generated assembly looks like.



pwn-hyperion

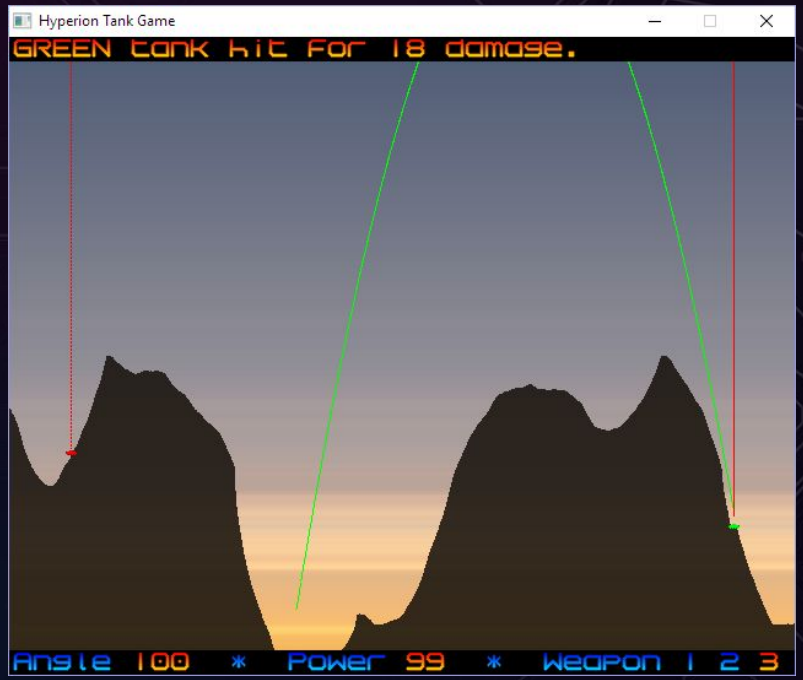
```
fprintf(stderr, "Works\n");
```

and unexpected stderr redirection...

Discrepancy in execution

VS

2. What the source code says will happen.
4. What the generated assembly looks like.




```
fprintf(stderr, "Works\n");
```

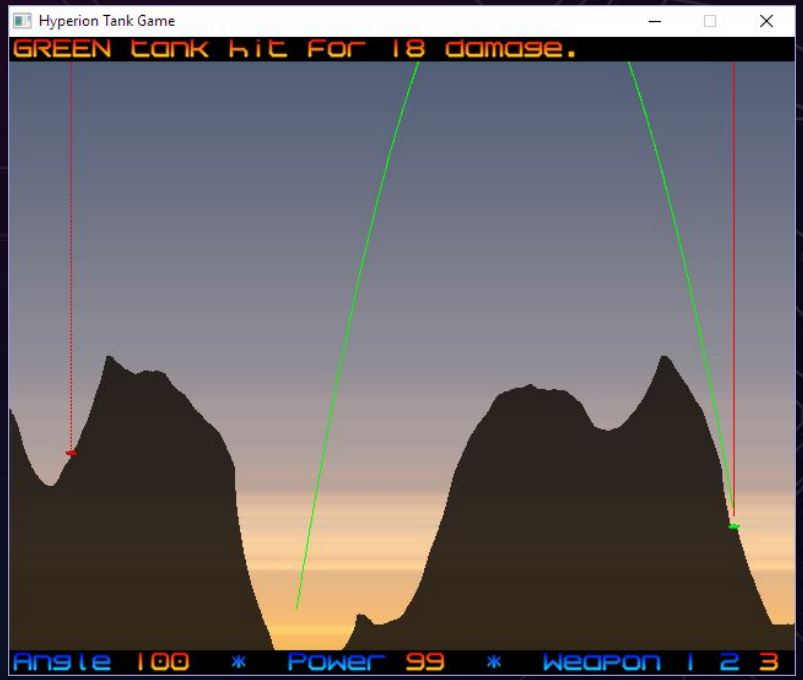
How to fix **without recompilation**?

Use a hex editor to change:
"Works\n" to "\0orks\n"

Discrepancy in execution

VS

2. What the source code says will happen.
4. What the generated assembly looks like.



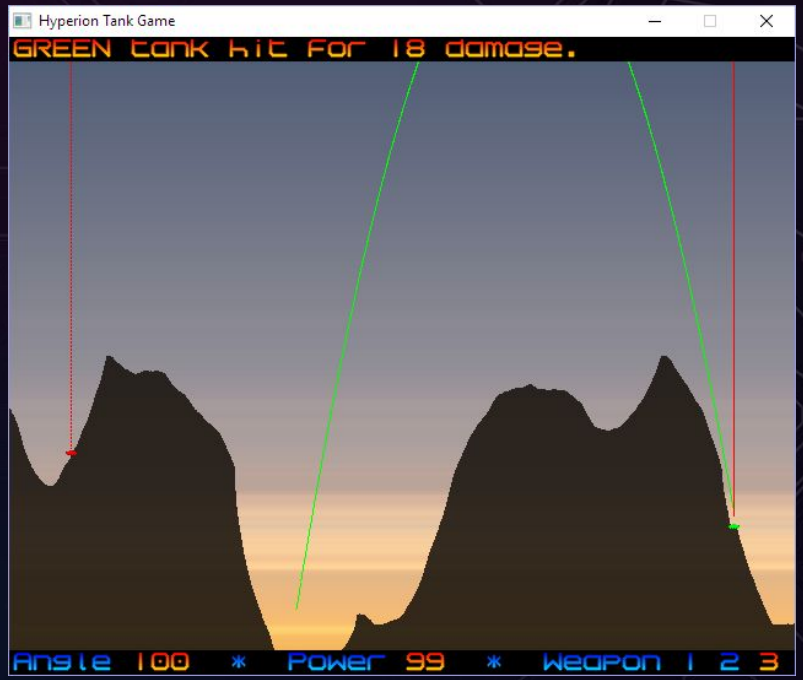

```
fprintf(stderr, "\0orks\n");
```

So... why does the server still send "\0orks\n"?

Discrepancy in execution

VS

2. What the source code says will happen.
4. What the generated assembly looks like.




```
fwrite("\0orks\n", 6, 1,
       stderr);
```

Fixed by NOPing out fwrite call.

Discrepancy in execution

VS

4. What the generated assembly looks like.
5. What the actual (dis)assembly looks like.

Discrepancy in execution

VS

4. What the generated assembly looks like.
5. What the actual (dis)assembly looks like.

```
int global;

extern "C" void start(void) {
  global = 12;
}
```

in all honesty
this kind of stuff
happens only if
you're already
doing weird stuff xD

Discrepancy in execution

VS

4. What the generated assembly looks like.
5. What the actual (dis)assembly looks like.

g++ -nostdlib -pie -fPIC -Wl,-estart pie.cc -o pie.s -S -masm=intel

```
mov    rax, QWORD PTR global@GOTPCREL[rip]
mov    DWORD PTR [rax], 12
nop
```

Discrepancy in execution

VS

4. What the generated assembly looks like.
5. What the actual (dis)assembly looks like.

```
g++ -nostdlib -pie -fPIC -Wl,-estart pie.cc -o pie_elf
objdump -d -Mintel pie_elf > pie_elf.s
```

yes, the **linker** changes the compiled machine code

```
lea    rax, [rip+0x200d44]
mov    DWORD PTR [rax], 12
nop
```

Discrepancy in execution

VS

4. What the generated assembly looks like.
5. What the actual (dis)assembly looks like.

g++ -nostdlib -pie -fPIC -Wl,-estart pie.cc -o pie_bin -Wl,-Tscript.lds
ndisasm -b64 pie_bin > pie_bin.s

```
mov    rax, QWORD PTR [rel 0x50]
mov    DWORD PTR [rax], 12
nop
```

too bad this doesn't actually work for "binary" type (pie is not supported) (ask how I found out heh)

Discrepancy in execution

Programmers say:

"Source code is the ultimate documentation"

Discrepancy in execution

Programmers say:

"Source code is the ultimate documentation"

Hackers disagree:

"Disassembled code is the ultimate source of truth"

Discrepancy in execution

Programmers say:

"Source code is the ultimate documentation"

Hackers disagree:

"Disassembled code is the ultimate source of truth"

We were all wrong.

Discrepancy in execution

5. What the actual (dis)assembly looks like.
6. What the CPU actually executes.

Discrepancy in execution

VS

5. What the actual (dis)assembly looks like.
6. What the CPU actually executes.

Fast forward to 2017/2018:

Spectre / Meltdown

Jann Horn / Google Project Zero: Reading privileged memory with a side-channel

Discrepancy in execution

simplified

VS

5. What the actual (dis)assembly looks like.
6. What the CPU actually executes.


```
// uint8_t secret[] is:
//    secret_sz of public data
//    several bytes of secret data
int x;
void guarded_secret(int idx) {  // Priv. code.
  if (idx < secret_sz) {
    x = x ^ detector[secret[idx] * CACHE_PAGE];
  }
}
```

Discrepancy in execution

simplified

VS

5. What the actual (dis)assembly looks like.
6. What the CPU actually executes.

Two branches:

1. if condition met / branch taken
2. if condition not met / branch not taken

```
void guarded_secret(int idx) {  // Priv. code.
  if (idx < secret_sz) {
    x = x ^ detector[secret[idx] * CACHE_PAGE];
  }
}
```

Discrepancy in execution

simplified

VS

5. What the actual (dis)assembly looks like.
6. What the CPU actually executes.

If secret_sz is a cache miss, CPU might use **speculative execution** to execute the most probable next branch (based on historical data).

```
void guarded_secret(int idx) {  // Priv. code.
  if (idx < secret_sz) {
    x = x ^ detector[secret[idx] * CACHE_PAGE];
  }
}
```

Discrepancy in execution

simplified

VS

5. What the actual (dis)assembly looks like.
6. What the CPU actually executes.

Attacker can:

1. Train the CPU to always take the if branch.
2. Flush secret_sz from cache and set idx to out-of-bounds value.

So what - the result won't be committed anyway.

```
void guarded_secret(int idx) {  // Priv. code.
  if (idx < secret_sz) {
    x = x ^ detector[secret[idx] * CACHE_PAGE];
  }
}
```

Discrepancy in execution

simplified

VS

5. What the actual (dis)assembly looks like.
6. What the CPU actually executes.

   Yes. The result won't be committed.

```
void guarded_secret(int idx) {  // Priv. code.
  if (idx < secret_sz) {
    x = x ^ detector[secret[idx] * CACHE_PAGE];
  }
}
```

Discrepancy in execution

simplified

VS

5. What the actual (dis)assembly looks like.
6. What the CPU actually executes.

   Yes. The result won't be committed.

But any cache fetches of attacker-controlled detector array WILL be easily detectable by measuring access time.

```
void guarded_secret(int idx) {  // Priv. code.
  if (idx < secret_sz) {
    x = x ^ detector[secret[idx] * CACHE_PAGE];
  }
}
```

Discrepancy in execution

simplified

VS

5. What the actual (dis)assembly looks like.
6. What the CPU actually executes.

   E.g. if secret[secret_sz+5] is 'C'...


```
void guarded_secret(int idx) {  // Priv. code.
  if (idx < secret_sz) {
    x = x ^ detector[secret[idx] * CACHE_PAGE];
  }
}
```

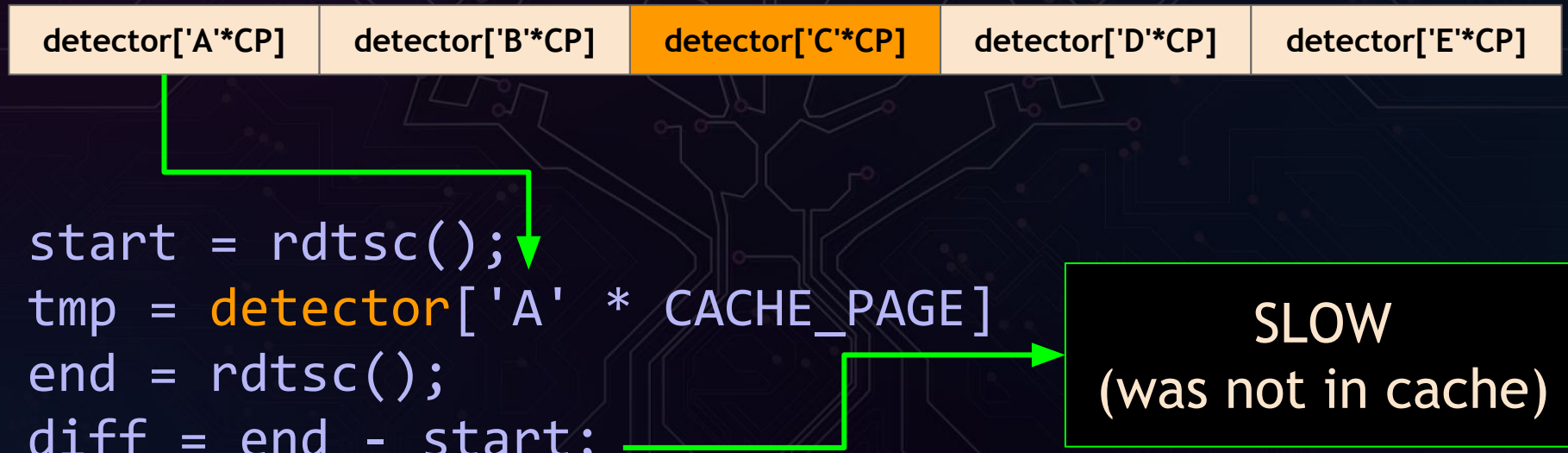
Discrepancy in execution

simplified

VS

5. What the actual (dis)assembly looks like.
6. What the CPU actually executes.

   E.g. if secret[secret_sz+5] is 'C'...



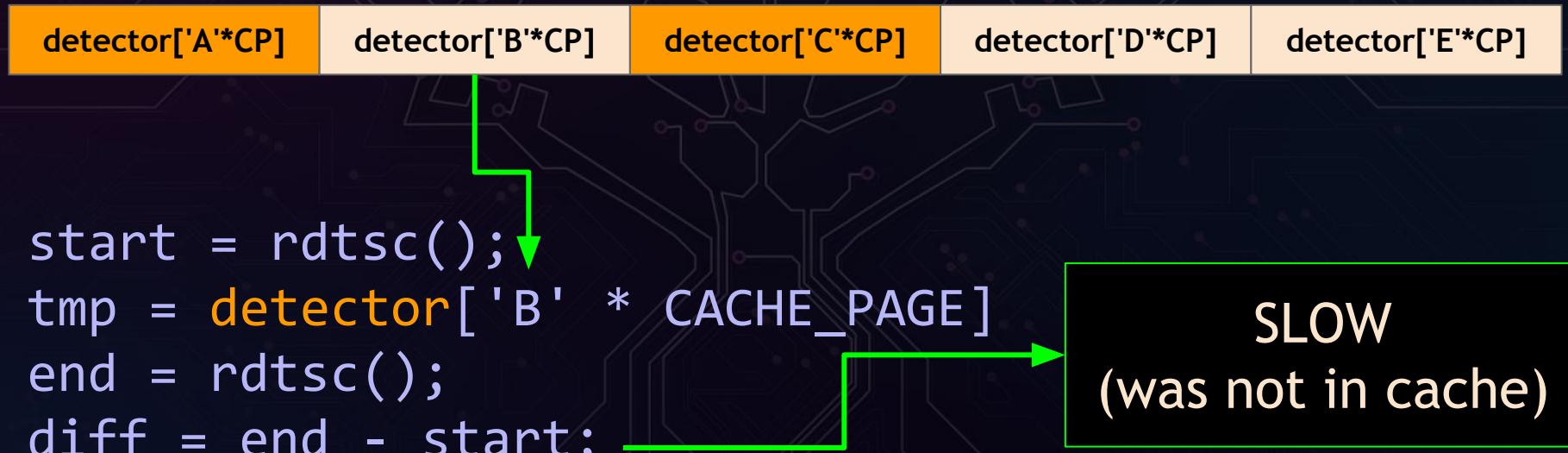
Discrepancy in execution

simplified

VS

5. What the actual (dis)assembly looks like.
6. What the CPU actually executes.

   E.g. if secret[secret_sz+5] is 'C'...



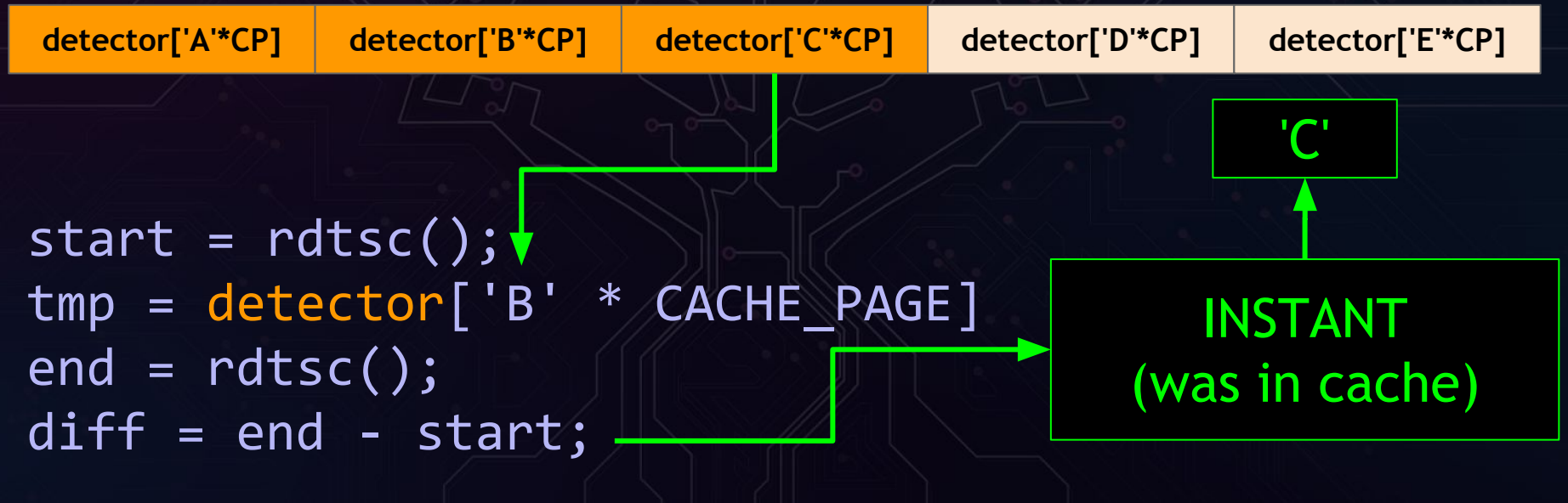
Discrepancy in execution

simplified

VS

5. What the actual (dis)assembly looks like.
6. What the CPU actually executes.

   E.g. if secret[secret_sz+5] is 'C'...



Discrepancy in execution

1. What the programmer meant.
2. What the source code says will happen.
3. What the standard says will happen.
4. What the generated assembly looks like.
5. What the actual (dis)assembly looks like.
6. What the CPU actually executes.

It's not easy to get everything right, even with proper code.

Other fun stuff: Info Leaks in padding.

```
struct {
  char a;
  // 3 bytes padding
  int b;
  char c;
  // another 3 bytes
};
```

```
long double x;
// sizeof(x) →
//    12 or 16
```

CPU register size: 10

(2-6 bytes of padding)

Other fun stuff: Memory ordering.

```
struct Data {
  int x, y;  // Data
  bool ready;
};
```

Thread 1:

```
x = 5; y = 7;
ready = true;
```

Thread 2:

```
while (!ready) {}
// process x, y
```

Other fun stuff: NaN NaN NaN NaN... Batman!

```
float f = NaN;
data[(int)f]
```

(3D graphics)

```
std::vector<double> array;
// User fills array.
std::sort(array.begin(),
array.end());
```

(credit: Redford)

```
if (f >= 42.0) { A(); }
else if (f < 42.0) { B(); }
else { C(); /* ??? */ }
```

Other fun stuff: "as if" rule in math

```
int div7(int x) {
  return x / 7;
}
```


```
int div7(int x) {
  return
    ((x * -1840700269 + x) >> 2) -
    (x >> 31);
}
```

Other fun stuff: Preprocessor macros - the classic.

Example credit: Redford

```
while (pos < input_len)
{
    size_t chunk_len =
        min(rand() % 8 + 2, input_len - pos);
    print_data(input + pos, chunk_len);
    printf(" ");
    pos += chunk_len;
}
```

Turned out min wasn't std::min, but a macro from Windows.h

Other fun stuff: Implicit integer overflows

```
auto x = new VeryLargeObject[very_large_number];

// C99 variable length local arrays
int array[user_controlled];
```

Other fun stuff: Recursion

Turns out in some shared-memory scenarios this is exploitable.

```
int func(attacker_controlled_t attacker_controlled) {
  large_object obj;
  ...
  if (condition) {
    func(attacker_controlled);
  }
  ...
}
```

Other fun stuff: Standard Library isn't flawless.

e.g. std::random_device::entropy (credit: Redford)

Notes

This function is not fully implemented in some standard libraries. For example, LLVM libc++ always returns zero even though the device is non-deterministic. In comparison, Microsoft Visual C++ implementation always returns 32, and boost.random returns 10.

e.g. DNS resolver in glibc
https://isc.sans.edu/forums/diary/CVE20157547+Critical+Vulnerability+in+glibc+getaddrinfo/20737/

e.g. strncat :)

Other fun stuff: Typical security stuff

- Buffer overflows
- Use-after-free
- All forms of race conditions
- Double-frees
- Type confusions (now also with `auto`!)
- Uninitialized variables
- Mistaking assert for an if :)

Other fun stuff: And other math stuff...

Floats are fun in general!

Exercise in integers: How much is -12 % 5?
(hint: it's either 3 or -2, depends on the language)

1234 << 32 → ? (yes, this is UB)

```
int k = -12;
while (k != 0) { k >>= 1; }
// How many iterations?
```

Summary

C++ is hard.

(But at least it doesn't have eval() or == and === operators...)

Know it well and take care while coding :)

(also, use modern C++ without pointer arithmetic whenever possible)

THE END

Are there any easy questions?

(If there are only hard ones then I'm sorry, but we run out of time ;>)

E-mail: gynvael@coldwind.pl Twitter: @gynvael YT: GynvaelEN

Blog: http://gynvael.coldwind.pl/ (Soon also: http://gynvael.live)