DE LA RECHERCHE À L'INDUSTRIE

Overengineered : 1337 * crackme-100

Generated by machines for machines

Camille MOUGEY   Florent MONJALET

Commissariat à l'Énergie Atomique et aux Énergies alternatives
Direction des Applications Militaires

17 novembre 2017

www.cea.fr

Once Upon a Time

`xarkes: hey guys, why don’t you write the last step of this year’s challenge?`

(freely translated and edited)

Once Upon a Time

`xarkes: hey guys, why don’t you write the last step of this year’s challenge?`

(freely translated and edited)

Guidelines :

- Reverse challenge
- Last step ⇒ should be... ~~tedious~~ *challenging* !
- No guess

Once Upon a Time

`xarkes: hey guys, why don’t you write the last step of this year’s challenge?`

(freely translated and edited)

Guidelines :

- Reverse challenge
- Last step ⇒ should be... ~~tedious~~ *challenging* !
- No guess

Idea : force people to use tools because *it’s the future, bro*

- Focus on automation, not on efficient manual analysis
- Prevent trivial attacks
- Miasm should not be the only viable solution (tough one)
- There should be some hype at the end

More is more

Implementation :

- Loads of binaries (let's say 1337)
- 4 architectures : x86, x86_64, ARM, AARCH64
- 2 OS : Windows, Linux
- ARM and AARCH64 are linux only, and there are fewer of them (5 of each)
- Each binary is a different equation to solve
- Each binary has its own packer
- Validator is an unnecessary concurrent rust source code

Inspiration

Misc

- Inspired by the DefCon 2017 challenge
- Should not be solvable with `grep`
- We really hope it wasn’t...

```
objdump -M intel -d magic/* | grep -P ”cmp\s+rdi”\
      | grep -oP ”0x\w{1,2}” | xxd -r -p
objdump -M intel -d sorcery/* | grep -P ” 3\w{3}.*cmp\s+[ac]l”\
      | grep -oP ”0x\w{1,2}” | xxd -r -p
objdump -M intel -d alchemy/* | grep -P ” 4[012]\w{4}:.*cmp\s+r[ac]x,0x\w{2}$”\
      | grep -oP ”0x\w{1,2}” | xxd -r -p
objdump -M intel -d witchcraft/* | grep -P ”[add|sub|cmp]\s+rdi,0x”\
      | cut -c33-80 | sed ’s/ /,/’ | python parser.py
```

Source : https ://github.com/sinfocol/ctfs

Producing equations

Approach 1 : smart way

- Produce a function *f* with **one and only one** value *x* such that $f(x) = 0$
- Apply reversible transformation, expand, reduce, …
- Do it 1337 times

Producing equations

Approach 1 : smart way

- Produce a function *f* with **one and only one** value *x* such that $f(x) = 0$
- Apply reversible transformation, expand, reduce, …
- Do it 1337 times

Approach 2 : lazy way

- Brute-force random equations
- Ask a SMT solver for the *one and only one answer* constraint

Producing equations

Approach 1 : smart way

- Produce a function *f* with **one and only one** value *x* such that $f(x) = 0$
- Apply reversible transformation, expand, reduce, …
- Do it 1337 times

Approach 2 : lazy way

- Brute-force random equations
- Ask a SMT solver for the *one and only one answer* constraint
- $\longrightarrow$ we have a winner !
- Do it 1337 times

Producing equations

Implementation

- Operations in the 2^n bit world $\longrightarrow$ Miasm IR !
- Start with the input, apply random operations with random constants to produces intermediates variables

Producing equations

Implementation

- Operations in the 2^n bit world → Miasm IR !
- Start with the input, apply random operations with random constants to produces intermediates variables
- Mix these intermediate variables together with random operations

Producing equations

Implementation

- Operations in the 2^n bit world $\longrightarrow$ Miasm IR !
- Start with the input, apply random operations with random constants to produces intermediates variables
- Mix these intermediate variables together with random operations
- Evaluate the sum of all variables (final equation) with one random value

Producing equations

Implementation

- Operations in the 2^n bit world → Miasm IR !
- Start with the input, apply random operations with random constants to produces intermediates variables
- Mix these intermediate variables together with random operations
- Evaluate the sum of all variables (final equation) with one random value
- Ask z3 (through Miasm) if there is only one way of getting this result

Producing equations

Implementation

- Operations in the 2^n bit world $\longrightarrow$ Miasm IR !
- Start with the input, apply random operations with random constants to produces intermediates variables
- Mix these intermediate variables together with random operations
- Evaluate the sum of all variables (final equation) with one random value
- Ask z3 (through Miasm) if there is only one way of getting this result
- Save the input for later (expected input)

Producing equations

Implementation

- Operations in the 2^n bit world $\longrightarrow$ Miasm IR !
- Start with the input, apply random operations with random constants to produces intermediates variables
- Mix these intermediate variables together with random operations
- Evaluate the sum of all variables (final equation) with one random value
- Ask z3 (through Miasm) if there is only one way of getting this result
- Save the input for later (expected input)
- Translate to C (Miasm IR $\longrightarrow$ (unreadable) C)

Misc

Avoid common attacks

- Avoid brute-force : input is 64 bits
- Patterns are random to avoid “grep attack”
- Avoid too easy tracing : insert randoms checks to avoid full equation dumping in one run

Misc

Avoid common attacks

- Avoid brute-force : input is 64 bits
- Patterns are random to avoid “grep attack”
- Avoid too easy tracing : insert randoms checks to avoid full equation dumping in one run

```
uint64_t test(uint64_t x) {
    uint64_t var0, var1, var2, var3, var4, var5, var6, var7, var8, var9;
    var0 = (x^x);
    var1 = (0x2BECFB880A6B7B72+var0);
    var2 = (var1+0x620D004B294BA344);
    if ((var1 & 0x2040080405110022) != 0x2040080000010022) return -1;
    var3 = (var2+var0);
    var4 = (0x671F8D008D0800D|var3);
    var5 = (var3&0x6E67FB8012DA33A);
    var6 = (var2+(- var5));
    var7 = (var4|0xC98A8C805C4FF93C);
    var8 = (var6|var0);
    if ((var8 & 0x608100018209001) != 0x8000010001000) return -1;
    var9 = (0x27A81200F061A58B+(- var3));
    return x + var0 + var1 + var2 + var3 + var4 + var5 + var6 + var7 + var8 + var9 - 0x8738A051601EC7DE;
}
```

Multiple tools

Several tools could be used

- Only a few challenges on ARM / AARCH64 : do-able by hand
- No float, no (too) exotic opcodes, no loops, …
- (probably) suitable tools
  - Triton
  - Manticore
  - Angr
  - Miasm
  - …

Working methods (on Miasm)

- Symbolic execution with state splitting
- Dynamic Symbolic Execution
- Dependency Graph

Packers

Cross platform polymorphic packer (the dumb way)

- Take a pre-compiled equation function
- Python script generates a random packer for it
  - Has a list of inversible operation : xor/xor, rol/ror, +/-…

Packers

Cross platform polymorphic packer (the dumb way)

- Take a pre-compiled equation function
- Python script generates a random packer for it
  - Has a list of inversible operation : xor/xor, rol/ror, +/-…
  - Picks an operator size (1, 2, 4 or 8 bytes)

Packers

Cross platform polymorphic packer (the dumb way)

- Take a pre-compiled equation function
- Python script generates a random packer for it
  - Has a list of inversible operation : xor/xor, rol/ror, +/-…
  - Picks an operator size (1, 2, 4 or 8 bytes)
  - Generates a list of packing and corresponding unpacking operations

Packers

Cross platform polymorphic packer (the dumb way)

- Take a pre-compiled equation function
- Python script generates a random packer for it
  - Has a list of inversible operation : xor/xor, rol/ror, +/-…
  - Picks an operator size (1, 2, 4 or 8 bytes)
  - Generates a list of packing and corresponding unpacking operations
  - Generates an ad-hoc C unpacker as it packs the original binary code

Packers

Cross platform polymorphic packer (the dumb way)

- Take a pre-compiled equation function
- Python script generates a random packer for it
  - Has a list of inversible operation : xor/xor, rol/ror, +/-…
  - Picks an operator size (1, 2, 4 or 8 bytes)
  - Generates a list of packing and corresponding unpacking operations
  - Generates an ad-hoc C unpacker as it packs the original binary code
- The packer just `mmaps`, unpacks, `mprotects` and executes the equation code
- Also cleans up its mess (`bzero` and `munmap`), we're kind of doing quality dev here

Packers

```
void unpack(ptype *buf)
{
    ptype *from, *to, c;
    for (from = packed, to = buf; (char *)from < (char *)packed + packed_size;
            from++, to++) {
        c = *from;
        c = rotr(c, 0xe);
        c = rotr(c, 0x1);
        c = c + 0x4b1bc27c;
        c = c - 0x457bc3da;
        c = c - 0x1823cae2;
        c = rotr(c, 0x1d);
        c = c ^ 0xaa907f80;
        c = c - 0x40f0f8b5;
        [...]
        c = rotr(c, 0xd);
        c = rotl(c, 0x1e);
        *to = c;
    }
}
```

Packers

Built to be bypassed

- Each of the 1337 packers is different
- “Highly” obfuscated (O-LLVM with all options)
- Not meant for static analysis
- Simple bypass : breaking on `mprotect/VirtualProtect`

Build Chain and Quality Assessment

The build should be automated : for x86 and x86_64 Windows and Linux, as well as ARM and AARCH64 linux

- Packing and compilation for all targets

Build Chain and Quality Assessment

The build should be automated : for x86 and x86_64 Windows and Linux, as well as ARM and AARCH64 linux

- Packing and compilation for all targets
- Compilation for all targets **with O-LLVM**
- Test every equation binary (wine and qemu)

Build Chain and Quality Assessment

The build should be automated : for x86 and x86_64 Windows and Linux, as well as ARM and AARCH64 linux

- Packing and compilation for all targets
- Compilation for all targets **with O-LLVM**
- Test every equation binary (wine and qemu)

The process :

1. Generate a random equation in C and store its solution

Build Chain and Quality Assessment

The build should be automated : for x86 and x86_64 Windows and Linux, as well as ARM and AARCH64 linux

- Packing and compilation for all targets
- Compilation for all targets **with O-LLVM**
- Test every equation binary (wine and qemu)

The process :

1. Generate a random equation in C and store its solution
2. Compile and obfuscate slightly (O-LLVM instruction substitution)

Build Chain and Quality Assessment

The build should be automated : for x86 and x86_64 Windows and Linux, as well as ARM and AARCH64 linux

- Packing and compilation for all targets
- Compilation for all targets **with O-LLVM**
- Test every equation binary (wine and qemu)

The process :

1. Generate a random equation in C and store its solution
2. Compile and obfuscate slightly (O-LLVM instruction substitution)
3. Extract equation function

Build Chain and Quality Assessment

The build should be automated : for x86 and x86_64 Windows and Linux, as well as ARM and AARCH64 linux

- Packing and compilation for all targets
- Compilation for all targets **with O-LLVM**
- Test every equation binary (wine and qemu)

The process :

1. Generate a random equation in C and store its solution
2. Compile and obfuscate slightly (O-LLVM instruction substitution)
3. Extract equation function
4. Pack it and generate unpacker in C

Build Chain and Quality Assessment

The build should be automated : for x86 and x86_64 Windows and Linux, as well as ARM and AARCH64 linux

- Packing and compilation for all targets
- Compilation for all targets **with O-LLVM**
- Test every equation binary (wine and qemu)

The process :

1. Generate a random equation in C and store its solution
2. Compile and obfuscate slightly (O-LLVM instruction substitution)
3. Extract equation function
4. Pack it and generate unpacker in C
5. Strip and obfuscate heavily (O-LLVM bogus control flow, control flow flattening, instruction substitution)

Build Chain and Quality Assessment

The build should be automated : for x86 and x86_64 Windows and Linux, as well as ARM and AARCH64 linux

- Packing and compilation for all targets
- Compilation for all targets **with O-LLVM**
- Test every equation binary (wine and qemu)

The process :

1. Generate a random equation in C and store its solution
2. Compile and obfuscate slightly (O-LLVM instruction substitution)
3. Extract equation function
4. Pack it and generate unpacker in C
5. Strip and obfuscate heavily (O-LLVM bogus control flow, control flow flattening, instruction substitution)
6. Test that it works

Build Chain and Quality Assessment

The build should be automated : for x86 and x86_64 Windows and Linux, as well as ARM and AARCH64 linux

- Packing and compilation for all targets
- Compilation for all targets **with O-LLVM**
- Test every equation binary (wine and qemu)

The process :

1. Generate a random equation in C and store its solution
2. Compile and obfuscate slightly (O-LLVM instruction substitution)
3. Extract equation function
4. Pack it and generate unpacker in C
5. Strip and obfuscate heavily (O-LLVM bogus control flow, control flow flattening, instruction substitution)
6. Test that it works
7. Repeat 1337 times, of course

Build Chain and Quality Assessment

The build should be automated : for x86 and x86_64 Windows and Linux, as well as ARM and AARCH64 linux

- Packing and compilation for all targets
- Compilation for all targets **with O-LLVM**
- Test every equation binary (wine and qemu)

The process :

1. Generate a random equation in C and store its solution
2. Compile and obfuscate slightly (O-LLVM instruction substitution)
3. Extract equation function
4. Pack it and generate unpacker in C
5. Strip and obfuscate heavily (O-LLVM bogus control flow, control flow flattening, instruction substitution)
6. Test that it works
7. Repeat 1337 times, of course
8. Update the validator to suit the equations generated

Build Chain and Quality Assessment

The build should be automated : for x86 and x86_64 Windows and Linux, as well as ARM and AARCH64 linux

- Packing and compilation for all targets
- Compilation for all targets **with O-LLVM**
- Test every equation binary (wine and qemu)

The process :

1. Generate a random equation in C and store its solution
2. Compile and obfuscate slightly (O-LLVM instruction substitution)
3. Extract equation function
4. Pack it and generate unpacker in C
5. Strip and obfuscate heavily (O-LLVM bogus control flow, control flow flattening, instruction substitution)
6. Test that it works
7. Repeat 1337 times, of course
8. Update the validator to suit the equations generated
9. Compile and test validator

Harder is better

For GreHack 2018 (or maybe tonight ?)

- Loops in the equation
- Heavy equation obfuscation
- Anti-emulation tricks
- Anti symbolic execution tricks
  $a = b \Rightarrow$ `b=0; for (i=0;i<a;i++) b++`
- Rarely supported architectures (sh4, msp430…)
- …

Overengineered Validator

`Validation process:`

1. Read a file with all the equation solutions (64 bit hex numbers)
2. Xor all the numbers -> gives an encryption key
3. Decrypt the flag

Overengineered Validator

`Validation process:`

1. Read a file with all the equation solutions (64 bit hex numbers)
2. Xor all the numbers -> gives an encryption key
3. Decrypt the flag **with ChaCha20**

Overengineered Validator

`Validation process:`

1. Read a file with all the equation solutions (64 bit hex numbers)
2. **Hash (sha256)** each number
3. Xor all the **hashes** -> gives an encryption key
4. Decrypt the flag with ChaCha20

Overengineered Validator

`Validation process:`

1. Read a file with all the equation solutions (64 bit hex numbers)
2. Hash (sha256) each number **concurrently with a pool of threads**
3. Xor all the hashes **with a global lock** -> gives an encryption key
4. Decrypt the flag with ChaCha20

Overengineered Validator

Validation process:

1. Read a file with all the equation solutions (64 bit hex numbers)
2. Hash (sha256) each number concurrently with a pool of threads
3. Xor all the hashes **with atomic operations** -> gives an encryption key
4. Decrypt the flag with ChaCha20

Overengineered Validator

`Validation process:`

1. Read a file with all the equation solutions (64 bit hex numbers)
2. Hash (sha256) each number concurrently with a pool of threads
3. Xor all the hashes with atomic operations -> gives an encryption key
4. **Check that the “relaxed” ordering does not create bugs with ARM**
5. Decrypt the flag with ChaCha20

Overengineered Validator

`Validation process:`

1. Read a file with all the equation solutions (64 bit hex numbers)
2. **Ensure that we don’t malloc a buffer for every line of the file (zero copy parsing)**
3. Hash (sha256) each number concurrently with a pool of threads
4. Xor all the hashes with atomic operations -> gives an encryption key
5. Check that the “relaxed” ordering does not create bugs with ARM
6. Decrypt the flag with ChaCha20

Overengineered Validator

`Validation process:`

1. Read a file with all the equation solutions (64 bit hex numbers)
2. Ensure that we don’t malloc a buffer for every line of the file (zero copy parsing)
3. Hash (sha256) each number concurrently with a pool of threads
4. Xor all the hashes with atomic operations -> gives an encryption key
5. Check that the “relaxed” ordering does not create bugs with ARM
6. Decrypt the flag with ChaCha20
7. **Make sure we used a nightly only feature (atomic integers)**

Overengineered Validator

`Validation process:`

1. Read a file with all the equation solutions (64 bit hex numbers)
2. Ensure that we don’t malloc a buffer for every line of the file (zero copy parsing)
3. Hash (sha256) each number concurrently with a pool of threads
4. Xor all the hashes with atomic operations -> gives an encryption key
5. Check that the “relaxed” ordering does not create bugs with ARM
6. Decrypt the **unicode (not only ascii)** flag with ChaCha20
7. Make sure we used a nightly only feature (atomic integers)

Overengineered Validator

`Validation process:`

1. Read a file with all the equation solutions (64 bit hex numbers)
2. Ensure that we don’t malloc a buffer for every line of the file (zero copy parsing)
3. Hash (sha256) each number concurrently with a pool of threads
4. Xor all the hashes with atomic operations -> gives an encryption key
5. Check that the “relaxed” ordering does not create bugs with ARM
6. Decrypt the unicode (not only ascii) flag with ChaCha20
7. Make sure we used a nightly only feature (atomic integers)
8. **All of this in Rust, for safety and performance sakes**

Overengineered Validator

Last step : enjoy refreshing the scoreboard.

Thank you!

Commissariat à l’énergie atomique et aux énergies alternatives
Centre de Bruyères-le-Châtel | 91297 Arpajon Cedex
T. +33 (0)1 69 26 40 00 | F. +33 (0)1 69 26 40 00
Établissement public à caractère industriel et commercial
RCS Paris B 775 685 019

CEA/DAM