Deobfuscation and beyond

Vasily Bukasov
and
Dmitry Schelkunov

https://re-crypt.com
Agenda

• We'll speak about obfuscation techniques which commercial (and not only) obfuscators use and how symbolic equation systems could help to deobfuscate such transformations
• We'll form the requirements for these systems
• We'll briefly skim over design of our mini-symbolic equation system and show the results of deobfuscation (and not only) using it
Software obfuscation

Is used for software protection against computer piracy

Is used for malware protection against signature-based and heuristic-based antiviruses
Common obfuscation techniques

Code virtualization

Do it over and over again...

Could be deobfuscated by common compiler theory algorithms

Fetch byte-code and arguments

Big switch 😊

Execute byte-code handler and process instruction

Save results
Common obfuscation techniques

Recursive substitution

XOR EAX, EBX

PUSH EBX
NOT [ESP]
OR [ESP], EAX
NOT [ESP]
PUSH EBX
NOT [ESP]
AND [ESP], EAX
POP EAX
OR EAX, [ESP]
ADD ESP, 4

Could be
deobfuscated by reverse recursive substitution

But ... we prefer generic solutions 😊
Common obfuscation techniques

<table>
<thead>
<tr>
<th>Opaque predicates</th>
<th>Garbage code</th>
</tr>
</thead>
<tbody>
<tr>
<td>PUSH EAX</td>
<td>JCC LABEL1</td>
</tr>
<tr>
<td>PUSH EAX</td>
<td>; Meaningless block start</td>
</tr>
<tr>
<td>ADD [ESP], EBX</td>
<td>PUSH EBX</td>
</tr>
<tr>
<td>PUSH EAX</td>
<td>NOT [ESP]</td>
</tr>
<tr>
<td>AND [ESP], EBX</td>
<td>AND [ESP], EAX</td>
</tr>
<tr>
<td>SHL [ESP], 1</td>
<td>AND EAX, EBX</td>
</tr>
<tr>
<td>POP EAX</td>
<td>PUSH EAX</td>
</tr>
<tr>
<td>SUB [ESP], EAX</td>
<td>AND EAX, [ESP + 4]</td>
</tr>
<tr>
<td>XOR [ESP], EBX</td>
<td>XOR [ESP], EAX</td>
</tr>
<tr>
<td>POP EAX</td>
<td>POP EAX</td>
</tr>
<tr>
<td>XOR [ESP], EAX</td>
<td>XOR [ESP], EAX</td>
</tr>
<tr>
<td>POP EAX</td>
<td>POP EAX</td>
</tr>
<tr>
<td>JZ LABEL1</td>
<td>XOR EAX, 0xFFFFFFF</td>
</tr>
<tr>
<td></td>
<td>; Meaningless block end</td>
</tr>
<tr>
<td>...</td>
<td>LABEL1:</td>
</tr>
<tr>
<td>&lt;DEAD CODE&gt;</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>LABEL1:</td>
<td>...</td>
</tr>
</tbody>
</table>
Common obfuscation techniques

Code duplication

OR EAX, EBX

JCC LABEL1

LABEL1:
PUSH EAX
ADD [ESP], EBX
PUSH EAX
AND [ESP], EBX
POP EAX
SUB [ESP], EAX
POP EAX

LABEL2:

...
Common obfuscation techniques

Code duplication in virtualization obfuscators

Instruction handler (before)

Instruction handler (after)

Very strong obfuscation technique but ... not in this case 😊
We could choose a branch randomly and forget about the rest 😊
Previous researches and products

• The Case for Semantics-Based Methods in Reverse Engineering, Rolf Rolles, RECON 2012


• CodeDoctor
  – deobfuscates simple expressions
  – plugin for OllyDbg and IDA Pro
Previous researches and products

• VMSweeper
  – declares deobfuscation (devirtualization) of Code Virtualizer/CISC and VMProtect (works well on about 30% of virtualized samples)
  – not a generic tool (heavily relies on templates)
  – works as a decompiler not optimizer
  – weak symbolic equation system

• CodeUnvirtualizer
  – declares deobfuscation (devirtualization) of Code Virtualizer/CISC/RISC and Themida new VMs
  – not a generic tool (heavily relies on templates)
  – no symbolic equation system
Previous researches and products

• Ariadne
  – complex toolset for deobfuscation and data flow analysis
  – includes a lot of optimization algorithms from compiler theory
  – no symbolic equation system
  – it seems to be dead 😞

• LLVM forks
  – are based on LLVM optimization algorithms (classical compiler theory algorithms)
  – we couldn’t find any decently working version
  – are limited by LLVM architecture (How fast LLVM works with 500 000 IR instructions? How much system resources it requires?)
The problem

Existing deobfuscation solutions are mostly based on classical compiler theory algorithms and too weak against modern obfuscators in the most of cases.
Solution

- Use symbolic equation system (SES) for deobfuscation
- Form input data for SES (translate source IR code to SES representation)
- Simplify expressions using SES
- Translate results from SES representation to IR
- Apply other deobfuscation transformations
Symbolic equation system

Should work with mixed expressions

\[ \text{0xff00} \& ( \text{0x100} + ( \text{0xff00} \& \text{a} ) ) = \text{0xff00} \& ( \text{0x100} + \text{a} ) \]

\[ \text{0xff} \& ( \text{a} \& \text{0xff} + \text{b} ) = \text{0xff} \& ( \text{a} + \text{b} ) \]

\[ \text{0xfffffffffe} \& \text{a} + 1 \& \text{a} = \text{0xfffffffff} \& \text{a} \]
Symbolic equation system

Should work with variables of various size

\[
\begin{align*}
al = 1 & \implies \text{eax}.1 = \text{eax}.0 & 0xfffff00 \land 1 \\
al = \text{bl} & \implies \text{eax}.1 = \text{eax}.0 & 0xfffff00 \land 0xff \land \text{ebx}.0 \\
ah = \text{bl} & \implies \text{eax}.1 = \text{eax}.0 & 0xffff00ff \land ( ( 0xff \land \text{ebx}.0 ) \ll 8 )
\end{align*}
\]
Symbolic equation system

Should preserve additional information about variables and constants

\[ \text{eax.1} = \text{eax.0} + \text{0xb19b00b5} \]

\[ \text{ebx.1} = \text{ebx.0} + \text{0xdeadbeef} \]
Symbolic equation system

Should optimize expressions like following

\[ eax.1 = ( eax.0 \land ebx.0 ) + ( (eax.0 \land ebx.0 ) \ll 1 ) = \]

\[ eax.1 = ( eax.0 \lor ebx.0 ) + ( eax.0 \land ebx.0 ) = \]
Symbolic equation system

Should optimize expressions like following

\[ \text{eax.1} = ( \text{eax.0} \ ^\lor \text{ebx.0} ) + ( (\text{eax.0} \ & \text{ebx.0} ) \ll 1 ) = \text{eax.0} + \text{ebx.0} \]

\[ \text{eax.1} = ( \text{eax.0} \ | \text{ebx.0} ) + ( \text{eax.0} \ & \text{ebx.0} ) = \text{eax.0} + \text{ebx.0} \]

\textbf{z3 returns something strange here 😊}
Unfortunately, we couldn’t find an appropriate third-party symbolic equation system engine and … we decided to create a new one for ourselves.

We called it Project Eq.
Eq design

eax.1 = ( ( eax.0 * 0xffffffff ) + 0xffffffff ) ^ 0xffffffff
Eq design

eax.1 = ((eax.0 * 0xffffffff) + 0xffffffff) ^ 0xffffffff
eax.1 = ( ( eax.0 * 0xffffffff ) + 0xffffffff ) ^ 0xffffffff
eax.1 = ( ( eax.0 * 0xffffffff ) + 0xffffffff ) ^ 0xffffffff
Eq design

eax.1 = ((eax.0 * 0xffffffff) + 0xffffffff) ^ 0xffffffff
Eq design

eax.1 = ( ( eax.0 * 0xffffffff ) + 0xffffffff ) ^ 0xffffffff
Eq design

eax.1 = ( ( eax.0 * 0xffffffff ) + 0xffffffff ) ^ 0xffffffff
Eq design

eax.1 = ( ( eax.0 * 0xffffffff ) + 0xffffffff ) ^ 0xffffffff

eax.0 (v)

eax.1 = eax.0

Profit! 😊
Eq design

We are too lazy 😊
To represent boolean expressions we use Zhegalkin form:

\[ e_{ax} | e_{bx} = e_{ax} \lor e_{bx} \land e_{ax} \land e_{bx} \]

This “trick” allows us to use same optimization primitives for boolean and arithmetic expressions
union rebx_type
{
    UINT32 rebx;
    WORD rbx;
    BYTE rblow[2];
};

void vmp_constant_playing(rebx_type &rebx)
{
    BYTE var0;
    union var1_type
    {
        UINT32 var;
        WORD var_med;
        BYTE var_low;
    } var1;

    var0 = rebx.rblow[0];
    rebx.rblow[0] = 0xe7;
    var1.var_med = rebx.rbx;
    var1.var_low = 0x18;
    rebx.rbx = var1.var_med;
    rebx.rblow[0] = var0;
}
Eq in work

Eq representation of the previous code (before optimization)

```
rebx.rebx.1 = ((0xff & rebx.rebx.0) ^ (0xffffffff00 & ((0xffff & 0xffff & ((0xff & 0x18) ^ (0xffffffff00 & ((0xffff & 0xffff & ((0xff & 0xe7) ^ (0xffffffff00 & rebx.rebx.0)) ^ (0xffff0000 & var1.0)))))) ^ (0xffff0000 & ((0xff & 0xe7) ^ (0xffffffff00 & rebx.rebx.0))))))
```
Eq in work

Eq representation of the previous code (after optimization)

rebx.rebx.1 = ( ( 0xff & rebx.rebx.0 ) ^ ( 0xffffffff00 & ( ( 0xffff & 0xffffff & ( ( 0xff & 0x18 ) ^ ( 0xffffffff00 & ( ( 0xffff & 0xffff & ( ( 0xff & 0xe7 ) ^ ( 0xffffffff00 & rebx.rebx.0 ) ) ) ) ) ) ) ) ) ^ ( 0xffffffff0000 & var1.0 ) ) ) ) ) ^ ( 0xffffffff0000 & ( ( 0xff & 0xe7 ) ^ ( 0xffffffff00 & rebx.rebx.0 ) ) ) ) ) ) ) = rebx.rebx.0

Profit! 😊
void rustock_sample(UINT32 &rebp, UINT32 &redi, UINT32 &resi)
{
    UINT32 var0, var1, var2;

    var0 = rebp;
    rebp = redi | rebp;
    var1 = redi & var0;
    resi = ~var1;
    var2 = rebp & resi;
    redi = var0 ^ var2;
}
Eq representation of the previous code (before optimization)

```plaintext
redi.1 = ( rebp.0 ^ ( ( ( rebp.0 & redi.0 ) ^ rebp.0 ^ redi.0 ) & ( 0xffffffff ^ ( rebp.0 & redi.0 ) ) ) )
```
Eq in work

Eq representation of the previous code (after optimization)

\[
\text{redi.1 = ( rebp.0 ^ ( ( rebp.0 & redi.0 ) ^ rebp.0 ^ redi.0 ) ) & ( 0xffffffff ^ ( rebp.0 & redi.0 ) ) ) } = \text{redi.0}
\]

Profit! 😊
Deobfuscation with Eq

Test function

```
PUSH EBP
MOV EBP,ESP
PUSH ESI
PUSH EDI
PUSH ECX
MOV ESI, DWORD PTR SS:[EBP+C]
MOV EDI, DWORD PTR SS:[EBP+8]
MOV ECX, DWORD PTR SS:[EBP+10]

LOOP_LABEL:
  OR ECX, ECX
  JE LABEL_EXIT

LABEL_EXIT:
  POP ECX
  POP EDI
  POP ESI
  MOV ESP, EBP
  POP EBP
  MOV EAX, 2
  RET

MOV AL, BYTE PTR DS:[ESI]
INC ESI
MOV BYTE PTR DS:[EDI], AL
INC EDI
DEC ECX
JMP SHORT LOOP_LABEL
```
Deobfuscation with Eq

After code virtualization
Deobfuscation with Eq

```c
UINT32 var0, var1, var2, var3, var4, var5, var6, var7, var8;

    var7 = resp + 0x8;
    var8 = resp;
    resp = resp - 0x10;
    var4 = (UINT32_PTR)*var7;
    var1 = resp + 0x14;
    var5 = (UINT32_PTR)*var1;
    var2 = resp + 0x1c;
    var6 = (UINT32_PTR)*var2;

label1:
    resp = resp - 0x8;
    if( var6 != 0x0 ) goto label2;
    reax = 0x2;
    var0 = (UINT32_PTR)*var8;
    resp = var8 + 0x4;
    goto var0;

label2:
    var3 = var4;
    var4 = var4 + 0x1;
    (UINT8_PTR)*var5 = (UINT8_PTR)*var3;
    var5 = var5 + 0x1;
    var6 = var6 - 0x1;
    resp = resp + 0x8;
    goto label1;
The result of deobfuscation
```
Deobfuscation with Eq

- ASProtect
- CodeVirtualizer/Themida/WinLicense
  - old CISC/RISC
  - new Fish/Tiger
- ExeCryptor
- NoobyProtect/SafeEngine
- Tages
- VMProtect
- Some others...

Were deobfuscated successfully 😊
Deobfuscation with Eq
Some numbers

<table>
<thead>
<tr>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instructions initially</td>
<td>~100</td>
</tr>
<tr>
<td>Instructions after obfuscation</td>
<td>~300 000</td>
</tr>
<tr>
<td>Instructions after deobfuscation</td>
<td>~200</td>
</tr>
<tr>
<td>Code generation time</td>
<td>~4 min</td>
</tr>
<tr>
<td>Code deobfuscation time</td>
<td>~2 min</td>
</tr>
<tr>
<td>Memory</td>
<td>~300 Mb</td>
</tr>
</tbody>
</table>
Obfuscation with Eq

We could use optimization not for deobfuscation only. What if we could stop optimization process at random step?
Obfuscation with Eq

bswap eax
bswap eax

Initial expression
(could be deobfuscated by reverse recursive substitution)

reax.3 = (( (( (( 0xff0000 & reax.0 ) >> 0x8 ) & 0xff0000 ) ^ ( ( reax.0 >> 0x18 ) & 0xff0000 ) ^ ( reax.0 & 0x18 ) & 0xff0000 ) ^ ( ( 0xff0000 & reax.0 ) << 0x8 ) & 0xff0000 ) ) >> 0x8 ) ^ ( ( ( 0xff0000 & reax.0 ) >> 0x8 ) ^ ( reax.0 >> 0x18 ) ^ ( reax.0 & 0x18 ) ^ ( ( 0xff00 & reax.0 ) << 0x8 ) ) >> 0x18 ) ^ ( ( ( 0xff0000 & reax.0 ) >> 0x8 ) ^ ( reax.0 & 0x18 ) ^ ( reax.0 & 0x18 ) ^ ( ( 0xff00 & ( ( 0xff0000 & reax.0 ) >> 0x8 ) ) << 0x8 ) ) ^ ( ( 0xff00 & reax.0 ) << 0x8 ) ) << 0x8 ) )
Obfuscation with Eq

Partially optimized expression looks like this:

```plaintext
reax.3 = ( ( 0xffff & reax.0 ) ^ ( ( 0xff0000 & ( reax.0 << 0x8 ) ) << 0x18 ) ^ ( ( 0xff00 & ( reax.0 >> 0x8 ) ) << 0x18 ) ^ ( ( reax.0 >> 0x18 ) << 0x18 ) ^ ( ( reax.0 << 0x18 ) << 0x18 ) ^ ( ( 0xff00 & ( 0xff0000 & ( reax.0 << 0x8 ) ) ) ^ ( 0xff00 & ( reax.0 >> 0x8 ) ) ^ ( reax.0 >> 0x18 ) ^ ( reax.0 << 0x18 ) ) ) ) << 0x8 )
```
Obfuscation with Eq

Partially optimized expression looks like this:

```c
reax.3 = ( ( 0xffff & reax.0 ) ^ ( ( 0xff0000 & ( reax.0 << 0x8 ) ) << 0x18 ) ^ ( ( 0xff00 & ( reax.0 >> 0x8 ) ) << 0x18 ) ^ ( ( reax.0 >> 0x18 ) << 0x18 ) ^ ( ( reax.0 << 0x18 ) << 0x18 ) ) ^ ( ( 0xff00 & ( ( 0xff0000 & ( reax.0 << 0x8 ) ) ^ ( 0xff00 & ( reax.0 >> 0x8 ) ) ) ^ ( reax.0 >> 0x18 ) ) ^ ( reax.0 << 0x18 ) ) ) << 0x8 )
```

or like this:

```c
reax.3 = ( ( 0xffff & reax.0 ) ^ ( 0x0 xor ( ( 0xff00 & ( reax.0 >> 0x8 ) ) << 0x18 ) ^ ( ( reax.0 >> 0x18 ) << 0x18 ) ^ ( ( reax.0 << 0x18 ) << 0x18 ) ) ^ ( ( 0xff00 & ( ( 0xff0000 & ( reax.0 << 0x8 ) ) ^ ( 0xff00 & ( reax.0 >> 0x8 ) ) ) ^ ( reax.0 >> 0x18 ) ) ^ ( reax.0 << 0x18 ) ) ) << 0x8 )
```
Obfuscation with Eq

- Easy to implement
- Hard to deobfuscate using classical compiler theory optimization algorithms
- Hard to deobfuscate using reverse recursive substitution
- No templates and signatures in the obfuscated code
Obfuscation with Eq

But this tricky obfuscation is still weak. It’s possible to deobfuscate these expressions using Eq project or another symbolic equation system.

And we have to go deeper!
Obfuscation with Eq

S-boxing of boolean function

reax.1 = reax.0 ^ rebx.0 ^ <tricky_const> ^ <tricky_const>

2 bits from reax.0  2 bits from rebx.0

S-box

2 bits from <tricky_const> are embedded to S-box

2 bits of the result
Obfuscation with Eq

Profit! 😊
Perspectives

- Obfuscation becomes stronger
  - Complex mathematical expressions are used more frequently
  - Merges with cryptography

- Obfuscation migrates to dark side
  - Protectors are dying
  - Malware market is growing
Perspectives

• Obfuscation becomes undetectable
  – Mimicry methods are improved
  – Obfuscators try to avoid method of recursive substitutions
  – Obfuscators use well-known high-level platforms

• LLVM becomes a generic platform for creating obfuscators
Questions