Cryptographic Function Identification in Obfuscated Binary Programs

Hackito Ergo Sum 2012
Joan Calvet – j04n.calvet@gmail.com
Presentation Outline

- Introduction to the Problem
- Proposed Solution
- Examples
- What’s Next?
INTRODUCTION TO THE PROBLEM
What's this?
What’s this?
Tiny Encryption Algorithm - Wikipedia, the free encyclopedia

en.wikipedia.org/wiki/Tiny_Encryption_Algorithm - Traduire cette page

The magic constant, 2654435769 or 9E3779B9 is chosen to be 232/φ, where φ is the golden ratio. TEA has a few weaknesses. Most notably, it suffers from ...  

Properties - Versions - Reference code - See also

You avez consulté cette page 15 fois. Dernière visite : 07/04/12

The RC5 Encryption Algorithm?  
www engr uconn edu/~zshi/...rc5.pdf - États-Unis - Traduire cette page

Format de fichier: PDF/Adobe Acrobat - Afficher
de RL Rivest - Cité 827 fois - Autres articles
Q32 = 10011111000110111011100110111001 = 9e3779b9. P64 =
1011011111000010100010110001010001011011011001010011011011...

Vous avez consulté cette page le 05/04/12.

books.google.fr/books?isbn=0470852852...

Man Young Rhee - 2003 - Computers - 405 pages
... b7e15163 + 9e3779b9 = 5618cblic 5[2] = 5[1] + Q32 = 5618cblic + 9e3779b9 =

Changeset 329 – CrypTool 2.0
https://www.cryptool.org/trac/CrypTool2/changeset/329
28 May 2009 – The magic constant, 2654435769 (Decimal) or 9E3779B9 (Hex) is chosen to be (2^32 / phi) where phi is the golden ratio.</Run> ...
<table>
<thead>
<tr>
<th>Tools</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Crypto Searcher</td>
<td>“TEA”</td>
</tr>
<tr>
<td>Draca v0.5.7b</td>
<td>“TEA/RC5/RC6”</td>
</tr>
<tr>
<td>Findcrypt v2</td>
<td>Ø</td>
</tr>
<tr>
<td>Hash &amp; Crypto Detector v1.4</td>
<td>“TEA/XTEA/TEAN”</td>
</tr>
<tr>
<td>PEiD KANAL v2.92</td>
<td>“TEA/N, RC5, RC6”</td>
</tr>
<tr>
<td>Kerckhoffs</td>
<td>Ø</td>
</tr>
<tr>
<td>Signsrch 0.1.7</td>
<td>“TEA”</td>
</tr>
<tr>
<td>SnD Crypto Scanner v0.5b</td>
<td>Ø</td>
</tr>
<tr>
<td>Tools</td>
<td>Answer</td>
</tr>
<tr>
<td>-------------------------------------------</td>
<td>----------------------</td>
</tr>
<tr>
<td>Crypto Searcher</td>
<td>“TEA”</td>
</tr>
<tr>
<td>Draca v0.5.7b</td>
<td>“TEA/RC5/RC6”</td>
</tr>
<tr>
<td>Findcrypt v2</td>
<td>Ø</td>
</tr>
<tr>
<td>Hash &amp; Crypto Detector v1.4</td>
<td>“TEA/XTEA/TEAN”</td>
</tr>
<tr>
<td>PEiD KANAL v2.92</td>
<td>“TEA/N, RC5, RC6”</td>
</tr>
<tr>
<td>Kerckhoffs</td>
<td>Ø</td>
</tr>
<tr>
<td>Signsrch 0.1.7</td>
<td>“TEA”</td>
</tr>
<tr>
<td>SnD Crypto Scanner v0.5b</td>
<td>Ø</td>
</tr>
</tbody>
</table>

That’s indeed the Tiny Encryption Algorithm!
What about this one?
What about this one?

No particular constants
<table>
<thead>
<tr>
<th>Tools</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Crypto Searcher</td>
<td>Ø</td>
</tr>
<tr>
<td>Draca v0.5.7b</td>
<td>Ø</td>
</tr>
<tr>
<td>Findcrypt v2</td>
<td>Ø</td>
</tr>
<tr>
<td>Hash &amp; Crypto Detector v1.4</td>
<td>Ø</td>
</tr>
<tr>
<td>PEiD KANAL v2.92</td>
<td>Ø</td>
</tr>
<tr>
<td>Kerckhoffs</td>
<td>Ø</td>
</tr>
<tr>
<td>Signsrch 0.1.7</td>
<td>Ø</td>
</tr>
<tr>
<td>SnD Crypto Scanner v0.5b</td>
<td>Ø</td>
</tr>
<tr>
<td>Tools</td>
<td>Answer</td>
</tr>
<tr>
<td>--------------------------------------------</td>
<td>--------</td>
</tr>
<tr>
<td>Crypto Searcher</td>
<td>ø</td>
</tr>
<tr>
<td>Draca v0.5.7b</td>
<td>ø</td>
</tr>
<tr>
<td>Findcrypt v2</td>
<td>ø</td>
</tr>
<tr>
<td>Hash &amp; Crypto Detector v1.4</td>
<td>ø</td>
</tr>
<tr>
<td>PEiD KANAL v2.92</td>
<td>ø</td>
</tr>
<tr>
<td>Kerckhoffs</td>
<td>ø</td>
</tr>
<tr>
<td>Signsrch 0.1.7</td>
<td>ø</td>
</tr>
<tr>
<td>SnD Crypto Scanner v0.5b</td>
<td>ø</td>
</tr>
</tbody>
</table>

Sigh.. That was still TEA!
What Can We Do?

• How to recognize different TEA implementations in a more reliable way?

• Is there something such implementations have to share?
Input-Output Relationship

- For a key $K$ and an encrypted text $C$, any TEA implementation produces the same decrypted text $C'$. 
Input-Output Relationship

- For a key $K$ and an encrypted text $C$, *any* TEA implementation produces the *same* decrypted text $C'$.

Could we identify TEA implementations by using their I/O relationship?
Input-Output Relationship

- For a key $K$ and an encrypted text $C$, any TEA implementation produces the same decrypted text $C'$.

Could we identify TEA implementations by using their I/O relationship?

(or any other cipher)
PROPOSED SOLUTION
How To Use Input-Output Relationship?

- Let’s say $P$ is a program implementing an unknown cryptographic algorithm.
How To Use Input-Output Relationship?

- Let’s say P is a program implementing an unknown cryptographic algorithm.

- We can not *realistically* use the I/O relationship to prove that P implements a particular crypto algorithm *on any inputs*.
  
  *(too many input states to test!)*
How To Use Input-Output Relationship?

- Let’s say \( \mathbf{P} \) is a program implementing an unknown cryptographic algorithm.

- We can not \textit{realistically} use the I/O relationship to prove that \( \mathbf{P} \) implements a particular crypto algorithm on any inputs.

  \( \textit{too many input states to test!} \)

- But we can observe one particular \( \mathbf{P} \) execution and collect its input-output values...
For example:

```
0x42  P  0xCafeBabe
```

```
0xDEADBEEF
```
For example:

- Now imagine that when we execute a reference implementation of TEA with the key 0x42 and the input text 0xCAFEBABE, it produces 0xDEADBEEF.

What does it mean?
For example:

- Now imagine that when we execute a reference implementation of TEA with the key 0x42 and the input text 0xCafeBabe, it produces 0xDEADBEEF.

  What does it mean?

- It proves that $P$ implements TEA on these particular input values.
Final Goal

• We are going to prove that a particular program $P$ behaves like a known cryptographic algorithm during a particular execution.

• It means that we are not going to prove a general semantic equivalence between $P$ and a cryptographic algorithm.
Workflow

Given a program $P$:

- **Step 1:** Collect $P$ execution trace.

- **Step 2:** Extract possible cryptographic algorithms with their parameters from $P$ execution trace (here is the magic).

- **Step 3:** Identify these algorithms by comparing their I/O relationship with those of known algorithms.
STEP 1: COLLECT EXECUTION TRACE
## Execution Trace

- **Pin:** Dynamic Binary Instrumentation framework.

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Read Registers</th>
<th>Written Registers</th>
<th>Read Memory</th>
<th>Written Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>4012b3</td>
<td>push ebp</td>
<td>ebp 0012de28</td>
<td>esp 0012bd94</td>
<td></td>
<td>12bd94 0012de28</td>
</tr>
<tr>
<td>4012b4</td>
<td>mov ebp, esp</td>
<td>esp 0012bd94</td>
<td>ebp 0012bd94</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4012b6</td>
<td>push ebx</td>
<td>ebx 02f00010</td>
<td>esp 0012bd80</td>
<td></td>
<td>12bd80 2f00010</td>
</tr>
</tbody>
</table>

...
STEP 2: CRYPTOGRAPHIC ALGORITHM EXTRACTION
How To Find Crypto Code ? (1)

• Cryptographic code constitutes only a part of programs, we need a way to find it.

• As we want to play with *obfuscated* programs, IDA functions will not be enough...
In obfuscated programs, such things can happen:

Win32.Swizzor’s packer
How To Find Crypto Code ? (2)
How To Find Crypto Code ? (2)

- Cryptographic algorithms usually apply a *same treatment* on their input-output parameters.
How To Find Crypto Code ? (2)

• Cryptographic algorithms usually apply a same treatment on their input-output parameters.

• It makes loops a cryptographic code feature.
How To Find Crypto Code ? (2)

- Cryptographic algorithms usually apply a same treatment on their input-output parameters.

- It makes loops a cryptographic code feature.

- But there are loops everywhere, not only in crypto... What kind of loops are we looking for?
Loops?

$c = 18$

switch($c$)

*case 18:*
  * $c = 24$
  * ...*

*case 24:*
  * $c = 6$
  * ...*

*case 6:*
  * ...*

Mebroot state-machine
Loops?

Mebroot state-machine

Unrolling optimization

\[
\begin{align*}
c & = 18 \\
\text{switch}(c) \\
\text{case 18:} & \\
& \quad \ldots \\
& \quad c = 24 \\
& \quad \ldots \\
\text{case 24:} & \\
& \quad \ldots \\
& \quad c = 6 \\
& \quad \ldots \\
\text{case 6:} & \\
& \quad \ldots \\
\end{align*}
\]

\[
\begin{align*}
0x00 \ & \text{inc eax} \\
0x01 \ & \text{inc ebx} \\
0x02 \ & \text{mov [ebx], eax} \\
0x03 \ & \text{inc eax} \\
0x04 \ & \text{inc ebx} \\
0x05 \ & \text{mov [ebx], eax} \\
0x06 \ & \text{inc eax} \\
0x07 \ & \text{inc ebx} \\
0x08 \ & \text{mov [ebx], eax}
\end{align*}
\]
Loops?

Mebroot state-machine

Unrolling optimization

```plaintext
0x00  inc eax
0x01  inc ebx
0x02  mov [ebx], eax
0x03  inc eax
0x04  inc ebx
0x05  mov [ebx], eax
0x06  inc eax
0x07  inc ebx
0x08  mov [ebx], eax
```
Loooooops

• We look for the same operations applied repeatedly on a set of data.
Looooops

• We look for the same operations applied repeatedly on a set of data.

“A loop is the repetition of a same sequence of machine instructions at least two times.”

(This sequence of instructions is the loop body.)
### Execution Trace

<table>
<thead>
<tr>
<th>...</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>401325</td>
<td>add ebx, edi</td>
</tr>
<tr>
<td>401327</td>
<td>sub edx, ebx</td>
</tr>
<tr>
<td>401329</td>
<td>dec dword ptr [ebp+0xc]</td>
</tr>
<tr>
<td>40132c</td>
<td>jnz 0x401325</td>
</tr>
<tr>
<td>401325</td>
<td>add ebx, edi</td>
</tr>
<tr>
<td>401327</td>
<td>sub edx, ebx</td>
</tr>
<tr>
<td>401329</td>
<td>dec dword ptr [ebp+0xc]</td>
</tr>
<tr>
<td>40132c</td>
<td>jnz 0x401325</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
### Execution Trace

<table>
<thead>
<tr>
<th>Iteration 1</th>
<th>Iteration 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>401325</td>
<td>401325</td>
</tr>
<tr>
<td>401327</td>
<td>401327</td>
</tr>
<tr>
<td>401329</td>
<td>401329</td>
</tr>
<tr>
<td>40132c</td>
<td>40132c</td>
</tr>
<tr>
<td>401325</td>
<td>401325</td>
</tr>
<tr>
<td>401327</td>
<td>401327</td>
</tr>
<tr>
<td>401329</td>
<td>401329</td>
</tr>
<tr>
<td>40132c</td>
<td>40132c</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

- add ebx, edi
- add ebx, edi
- sub edx, ebx
- sub edx, ebx
- dec dword ptr [ebp+0xc]
- dec dword ptr [ebp+0xc]
- jnz 0x401325
- jnz 0x401325
- jnz 0x401325
- jnz 0x401325
## Example

### Execution Trace

<table>
<thead>
<tr>
<th>Address</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>401325</td>
<td>add ebx, edi</td>
</tr>
<tr>
<td>401327</td>
<td>sub edx, ebx</td>
</tr>
<tr>
<td>401329</td>
<td>dec dword ptr [ebp+0xc]</td>
</tr>
<tr>
<td>40132c</td>
<td>jnz 0x401325</td>
</tr>
<tr>
<td>401325</td>
<td>add ebx, edi</td>
</tr>
<tr>
<td>401327</td>
<td>sub edx, ebx</td>
</tr>
<tr>
<td>401329</td>
<td>dec dword ptr [ebp+0xc]</td>
</tr>
<tr>
<td>40132c</td>
<td>jnz 0x401325</td>
</tr>
</tbody>
</table>

### Loop

- **Iteration 1**
- **Iteration 2**
What About Nested Loops?

Simplified CFG
What About Nested Loops?

Simplified CFG

Execution trace
What About Nested Loops?

Simplified CFG

Execution trace

Loop B
3 iterations

Loop B
2 iterations

45
What About Nested Loops?

Simplified CFG

Execution trace
What About Nested Loops?

Simplified CFG

Execution trace

Different!
What About Nested Loops?

Simplified CFG

Execution trace
What About Nested Loops?

Simplified CFG

Execution trace

Trace
Rewriting

Ok!
What About Nested Loops?

Simplified CFG

Execution trace

Trace Rewriting

Ok!
Loop Detection Algorithm

1. Detects two repetitions of a loop body in the execution trace.
   *(non trivial, because there is an infinity of possible loop body)*

2. Replaces in the trace the detected loop by a symbol representing their body.

3. Goes back to step 1 if new loops have been detected.
What’s Next?

• We extracted possible cryptographic code from execution traces thanks to a particular loop definition.
What’s Next?

• We extracted possible cryptographic code from execution traces thanks to a particular loop definition.

• For the moment, we assume that each possible cryptographic algorithm corresponds to one single loop.
What’s Next?

• We extracted possible cryptographic code from execution traces thanks to a particular loop definition.

• For the moment, we assume that each possible cryptographic algorithm corresponds to one single loop.

• How can we define parameters from the bytes read and written in the execution trace?
Loop Parameters (1)

• Distinction between input and output bytes in the execution trace:
  
  – *Input* bytes have been *read without having been previously written*.

  – *Output* bytes have been *written*.
Loop Parameters (2)

• We want to group together bytes belonging to the same cryptographic parameter (key, input text...).
Loop Parameters (2)

• We want to group together bytes belonging to the same cryptographic parameter (key, input text...).

What criteria can we use?
Loop Parameters (3)

• Grouping of several bytes into the same parameter:
  1. If they are adjacent in memory *(too large!)*
Loop Parameters (3)

• Grouping of several bytes into the same parameter:
  1. If they are adjacent in memory *(too large!)*
  2. And if they are manipulated by a same instruction in the loop body.
Loop Parameters (3)

• Grouping of several bytes into the same parameter:
  1. If they are adjacent in memory (*too large!*)
  2. And if they are manipulated by a same instruction in the loop body.

```assembly
add ebx, edi  
mov eax, [ebx]  
...  
add ebx, edi  
mov eax, [ebx]  
...```

Loop Parameters (3)

• Grouping of several bytes into the same parameter:
  1. If they are adjacent in memory *(too large!)*
  2. And if they are manipulated by a same instruction in the loop body.

```assembly
Iteration 1
  add ebx, edi
  mov eax, [ebx]
  ...

Iteration 2
  add ebx, edi
  mov eax, [ebx]
  ...
```
Loop Parameters (3)

• Grouping of several bytes into the same parameter:
  1. If they are adjacent in memory *(too large!)*
  2. And if they are manipulated by a same instruction in the loop body.

```assembly
Iteration 1
- add ebx, edi
- mov eax, [ebx]
- ...
Iteration 2
- add ebx, edi
- mov eax, [ebx]
- ...
```
Loop Parameters (3)

• Grouping of several bytes into the same parameter:
  
  1. If they are **adjacent in memory** *(too large!)*
  
  2. And if they are **manipulated by a same instruction in the loop body.**

Iteration 1

- `add ebx, edi`
- `mov eax, [ebx]`
- ...

Iteration 2

- `add ebx, edi`
- `mov eax, [ebx]`
- ...

```
Loop Parameters (4)

- Once a parameter has been defined, we collect its value from the trace.
Loop Parameters (4)

- Once a parameter has been defined, we collect its value from the trace.

| Parameter | (memory address|register name):size | Value in hexadecimal |
|-----------|---------------------|----------------------|
Let’s Recap With a Use-Case

• Tiny Encryption Algorithm:
  – Block cipher
  – 64 round Feistel network
  – 16-byte key
  – 8-byte input text
  – Magic constant \textit{delta} (0x9E3779B9)

• We built a toy program calling the TEA decryption function on:
  – Key: \texttt{0xDEADBEE1...DEADBEE4}
  – Encrypted text: \texttt{0x0123456789ABCDEF}
Step 1: collect the execution trace
Step 2: cryptographic algorithm extraction
Step 2: cryptographic algorithm extraction
Step 2: cryptographic algorithm extraction
Step 2: cryptographic algorithm extraction
Step 2: cryptographic algorithm extraction
STEP 3: CRYPTO ALGORITHM IDENTIFICATION
**Input 1:** unknown algorithm A with its parameter values
**Input 1:** unknown algorithm $A$ with its parameter values

```
0x12ff44:4
01234567
deadbee3
0x12ff64:4
deadbee4
0x12ff68:4
89abcdef
deadbee1
0x12ff40:4
deadbee2
0x12ff5c:4
deadbee1
0x12ff60:4
deadbee2
0x12ff38:4
9e3779b9
0012ff4c
```

**Input 2:** reference implementations for common crypto algo

```python
def tea(input_text, key):
    ...

def xtea(input_text, key):
    ...

def rc4(input_text, key):
    ...
```
Question

• Is there a way to combine A input values such that tea(), xtea() or rc4() would produce A output values?
Question

• Is there a way to combine $A$ input values such that $tea()$, $xtea()$ or $rc4()$ would produce $A$ output values?

• Some difficulties:
  – **Parameter division**: a same cryptographic parameter can be divided into several loop parameter.
Question

• Is there a way to combine A input values such that \textit{tea()}, \textit{xtea()} or \textit{rc4()} would produce A output values?

• Some difficulties:
  – **Parameter division**: a same cryptographic parameter can be divided into several loop parameter.
  – **Parameter order**: no particular order for A parameters.
Question

• Is there a way to combine \( A \) input values such that \( \text{tea()} \), \( \text{xtea()} \) or \( \text{rc4()} \) would produce \( A \) output values?

• Some difficulties:
  – **Parameter division**: a same cryptographic parameter can be divided into several loop parameter.
  – **Parameter order**: no particular order for \( A \) parameters.
  – **Parameter number**: we collect more than the cryptographic parameters.
Brute-Force!
1. Generate all possible values with A input parameters:
   1. Length 4: 00000020, 01234567, deadbee3...
   2. Length 8: 0000002001234567, 000000020deadbee3,..
   3. ...
1. **Generate all possible values with A input parameters:**
   1. Length 4: `00000020, 01234567, deadbee3...`
   2. Length 8: `0000002001234567, 00000020deadbee3,..`
   3. ...

2. **Same thing with A output parameters.**
1. **Generate all possible values with A input parameters:**
   1. Length 4: \(00000020, 01234567, \text{deadbee3}...\)
   2. Length 8: \(0000002001234567, 00000020\text{deadbee3}...\)
   3. ...

2. **Same thing with A output parameters.**

3. **For TEA reference implementation:**
   1. Possible input texts (8 bytes): \(0000002001234567,...\)
1. Generate all possible values with A input parameters:
   1. Length 4: 00000020, 01234567, deadbee3...
   2. Length 8: 0000002001234567, 00000020deadbee3...
   3. ...

2. Same thing with A output parameters.

3. For TEA reference implementation:
   1. Possible input texts (8 bytes): 0000002001234567,...
   2. Possible keys (16 bytes): ...
1. Generate all possible values with A input parameters:
   1. Length 4: 00000020, 01234567, deadbee3...
   2. Length 8: 00000002001234567, 000000020deadbee3,..
   3. ...

2. Same thing with A output parameters.

3. For TEA reference implementation:
   1. Possible input texts (8 bytes): 00000002001234567,...
   2. Possible keys (16 bytes): ...
   3. Execute our TEA reference implementation on each possible pair (input text, key)
1. Generate all possible values with A input parameters:
   1. Length 4: 00000020, 01234567, deadbee3...
   2. Length 8: 00000002001234567, 000000020deadbee3, ...
   3. ...

2. Same thing with A output parameters.

3. For TEA reference implementation:
   1. Possible input texts (8 bytes): 00000002001234567,...
   2. Possible keys (16 bytes): ...
   3. Execute our TEA reference implementation on each possible pair (input text, key)
   4. If the output has been produced during step 2: success!
STARTED AT
2012-04-08 08:59:58.284000

\*\* Crypto Algorithm Identification starting !\*\*
9 input parameters
7 output parameters
All possible input values generation... Done!
All possible output values generation... Done!
Build internal structure... Done!
Comparison phase starting... Test for TEA decryption...

\*\* Found TEA decryption \*\*

>>> Key (16 bytes) : deadbee1deadbee2deadbee3deadbee4

>>> Crypted text (8 bytes) : 0123456789abcdef

>>> Decrypted text (8 bytes) : df5ec1536e089494

ENDED AT
2012-04-08 09:01:37.832000

\~ 2 minutes
Malware And TEA

EXAMPLES!
Storm Worm

• Several internet references about the use of TEA in the Storm Worm packer (aka Tibs).

• Let’s take a look to the code...
Let's try our tool...

Classic TEA operations

TEA delta

TEA round number
Storm Worm Sample → TRACER → Execution Trace → CRYPTO EXTRACTION → Unknown Algorithms
• For the previous loop, we extracted many unknown algorithms like these ones:
• For the previous loop, we extracted many unknown algorithms like these ones:
• For the previous loop, we extracted many unknown algorithms like these ones:

Looks like 8-byte cipher block (like TEA!) in ECB mode
Unknown Algorithms → IDENTIFICATION → ...
STARTED AT
2012-04-08 14:20:30.858000
*** Crypto Algorithm Identification starting !***
8 input parameters
5 output parameters
All possible input values generation... Done!
All possible output values generation... Done!
Build internal structure... Done!
Comparison phase starting... Test for TEA decryption... Unknown Algorithm stays unknown!
Done!
ENDED AT
2012-04-08 14:21:11.328000

WTF ?
Original TEA source code

\[ z = ((y << 4) + k[2]) \oplus (y + \text{sum}) \oplus ((y >> 5) + k[3]) \]

Storm Worm implementation

\[ z = 16 \times y + (y \times (\text{key} + 8)) + (\text{sum} \times (y >> 5)) + (\text{key} + 12) \]
Original TEA source code

\[ z := ((y \ll 4) + k[2]) \land (y + \text{sum}) \land ((y \gg 5) + k[3]) \]

Storm Worm implementation

\[ z := 16 \times y + (y \land (\text{key} + 8)) + (\text{sum} \land (y \gg 5)) + (\text{key} + 12) \]
Original TEA source code

\[ z = ((y \ll 4) + k[2]) \oplus (y + \text{sum}) \oplus ((y \gg 5) + k[3]) \]

Storm Worm implementation

\[ z = 16 \times y + (y \oplus (\text{key} + 8)) + (\text{sum} \oplus (y \gg 5)) + *(\text{key} + 12) \]
Original TEA source code

\[ z := \left( (y < 4) + k[2] \right)^\wedge (y + \text{sum})^\wedge \left( (y >> 5) + k[3] \right) \]

Storm Worm implementation

\[ z := 16 \times y + (y \wedge \times(\text{key} + 8)) + (\text{sum} \wedge (y >> 5)) + \times(\text{key} + 12) \]

This is not TEA: parenthesis at the wrong place!
Ok, Storm Worm implementation added to the base... (this is *not* TEA)
Trojan.SilentBanker

• Several internet references about the use of TEA in SilentBanker.

• Let’s take a look to the code...

(sounds familiar, isn’t it ?)
```assembly
loc_4012EC:
    mov    edi, [ebp+var_4]
    mov    eax, edx
    shr    eax, 5
    xor    eax, [ebp+arg_0]
    xor    edi, edx
    add    edi, [ebp+var_8]
    mov    ebx, edx
    shl    ebx, 4
    add    edi, eax
    add    ebx, edi
    mov    edi, [ebp+var_C]
    sub    ecx, ebx
    mov    eax, ecx
    xor    edi, ecx
    add    edi, [ebp+var_10]
    mov    ebx, ecx
    shr    eax, 5
    xor    eax, [ebp+arg_0]
    add    [ebp+arg_0], 61C88647h
    shl    ebx, 4
    add    edi, eax
    add    ebx, edi
    sub    edx, ebx
    dec    [ebp+arg_4]
    jnz    short loc_4012EC
```
Classic TEA operations

= sub [ebp+arg_0], 0x9E3779B9

TEA classic constant
(delta * round number)

TEA round number

mov [ebp+arg_0], 8C6EF3726h
mov [ebp+var_C], edi
mov [ebp+var_10], eax
mov [ebp+arg_4], 20h
Let’s try our tool...

Classic TEA operations

TEA round number

TEA classic constant

(delta * round number)

= sub [ebp+arg_0], 0x9E3779B9
• For the previous loop, we extracted many unknown algorithms like these ones:
• For the previous loop, we extracted many unknown algorithms like these ones:

Looks like 8-byte cipher block (like TEA!) in ECB mode
Fail.. Again !?
Same implementation than in the Storm Worm!
• They probably both copied/pasted a wrong source code from the internet.

• Started to look for it: Google, TEA Wikipedia page,... nothing!

• At some point, I remembered something these two malware families have in common...
They both came from Russia!
In cryptography, the Tiny Encryption Algorithm (TEA) is a very simple block cipher, implemented in only a few lines of code. It was designed by David Wheeler and Roger Needham at the Fast Software Encryption workshop in Leuven in 1994.

The cipher is not subject to any patents.

Properties

TEA operates on two 32-bit unsigned integers (could be bytes) a suggested 64 rounds, typically implemented in pairs of bytes, exactly the same way for each cycle. Different multiple of rounds. The magic constant, 2654435769 or 9E3779B9

TEA has a few weaknesses. Most notably, it suffers from a significant bias in the effective key size is only 126 bits.[3] As a result, TEA is unsuitable for modern cryptography. It requires 2^{23} chosen plaintexts under a related-key pair to be designed.
Ссылки

- Страница алгоритма шифрования TEA
- Roger M. Needham and David J. Wheeler. «TEA, a Tiny Encryption Algorithm» (PDF)
- Andrew Hang. «Hacking the Xbox: an introduction to reverse engineering»
- David J. Wheeler and Roger M. Needham. «Correction to xtea.» Technical report, Computer Laboratory, University of Cambridge
- Roger M. Needham and David J. Wheeler. «Tea extensions.» Technical report, Computer Laboratory, University of Cambridge
- Test vectors for TEA
- JavaScript implementation of XXTEA with Base64
Взлом как образ мысли

Интервью с человеком, который, как оказалось, является не только талантливым пентестером в одной из крупных ИБ-компании, но и хакером-вeterаном, который уверенными шагами вышел на свет и прикоснулся к истории российской хак-сцены....

ТЕА: блочный шифр своими руками

Дата: 22.04.2004 | slon

В данном тексте хотелось бы затронуть такую житотрепещущую тему, как шифрование файлов. Вообще нужно различать два вида шифрования файлов:

- шифрование для себя (чтобы ваши файлы никто, кроме вас не "читал")
- шифрование для других (чтобы ваши файлы "читал" только адресат)
push edi ; Сохраняем
mov ebx,v0 ; Кладём
mov ecx,v1 ; В ecx кл
mov edx,9e3779b9h ;
mov eax,edx ; Кладём
shl eax,5 ; Сдвиг eax
mov edi,32 ; Кладём DLoopR;
mov ebp,ebx ; Кладём
shl ebp,4 ; Сдвиг ebp
sub ecx,ebp ; Отнимаем
mov ebp,k2 ; Кладём
xor ebp,ebx ; XOR'им
sub ecx,ebp ; Отнимаем
mov ebp,ebx ; Кладём
shr ebp,5 ; Сдвиг ebp
xor ebp,eax ; XOR'им
sub ecx,ebp ; Отнимаем
sub ecx,k3 ; Отнимаем
;
mov ebp,ecx ; Кладём
shl ebp,4 ; Сдвиг ebp
sub ebx,ebp ; Отнимаем
mov ebp,k0 ; Кладём
xor ebp,ecx ; XOR'им
sub ebx,ebp ; Отнимаем
mov ebp,ecx ; Кладём
shr ebp,5 ; Сдвиг ebp
xor ebp,eax ; XOR'им
sub ebx,ebp ; Отнимаем
sub ebx,k1 ; Отнимаем
sub eax,edx ; Отнимаем
dec edi ; Уменьшаем
jnz DLoopR ; Делим
pop edi ; Вынимаем и
mov v0,ebx ; Кладём
mov v1,ecx ; В отведён
ret ; Возвращаем
push edi
mov ebx,v0 ; Кладём
mov ecx,v1 ; В ecx кл
mov edx,9e3779b9h ;
mov eax,edx ; Кладём
shl eax,5 ; Сдвиг eax
mov edi,32 ; Кладём
DLoopR:
mov ebp,ebx ; Кладём
shl ebp,4 ; Сдвиг ebp
sub ecx,ebp ; Отнимаем
mov ebp,k2 ; Кладём
xor ebp,ebx ; XOR'им
sub ecx,ebp ; Отнимаем
mov ebp,ebx ; Кладём
shr ebp,5 ; Сдвиг ebp
xor ebp,eax ; XOR'им
sub ecx,ebp ; Отнимаем
sub ecx,k3 ; Отнимаем
;
mov ebp,ecx ; Кладём
shl ebp,4 ; Сдвиг ebp
sub ebx,ebp ; Отнимаем
mov ebp,k0 ; Кладём
xor ebp,ecx ; XOR'им
sub ebx,ebp ; Отнимаем
mov ebp,ecx ; Кладём
shr ebp,5 ; Сдвиг ebp
xor ebp,eax ; XOR'им
sub ebx,ebp ; Отнимаем
sub ebx,k1 ; Отнимаем
sub eax,edx ; Отнимаем
dec edi ; Уменьшаем
jnz DLoopR ; Денифрировать

pop edi ; Вынимаем
mov v0,ebx ; Кладём
mov v1,ecx ; В отвод
ret ; Возврат из подпрограммы

loc_1FA1:
mov ebp, ebx
shl ebp, 4
sub ecx, ebp
mov ebp, [esi+8]
xor ebp, ebx
sub ecx, ebp
mov ebp, ebx
shr ebp, 5
xor ebp, eax
sub ecx, ebx
mov ebp, ecx
shl ebp, 4
sub ebx, ebp
mov ebp, [esi]
xor ebp, ecx
mov ebx, ebp
shr ebp, 5
xor ebp, eax
sub ebx, ebp
mov ebx, [esi+4]
sub eax, edx
dec edi
jnz short loc_1FA1

pop edi
mov [edi], ebx
mov [edi+4], ecx
ret

120
Modified TEA

MORE EXAMPLES!
Remember This?

The magic TEA constant \((delta)\) and the round number are seen as input parameters, because they are initialized \(\textit{before}\) the loop and used inside.
Modified TEA Implementation

- $\delta = 0x12345678$ \textit{(normally } 0x9E3779B9\textit{)}
- round number $= 16$ \textit{(normally 32)}
• TEA reference implementation extended:

```python
def tea(input_text, key):
    ...
```
• TEA reference implementation extended:

```python
def tea(input_text, key, delta, round_number):
    ...
```
TEA reference implementation extended:

```python
def tea(input_text, key, delta, round_number):
...
```
MORE EXAMPLES!
RC4 (1)

• RC4 algorithm:
  – Stream cipher
  – Variable-length key
  – Two loops generate a pseudorandom stream into a 256 bytes substitution-box (S-BOX).
  – A final loop does the actual decryption.

• We have to extend our model to regroup different loops into a same algorithm.
Interlude: Loop Data-Flow

• Two loops $L_1$ and $L_2$ are in the same algorithm:
  – If $L_1$ started before $L_2$ in the trace.
  – If $L_2$ uses as input parameter an output parameter of $L_1$.

(or the contrary!)
We built a toy program calling the RC4 decryption function on:
- Key: “SuperKeyIsASuperKey” (19 bytes)
- Encrypted text: “AAA....AA” (1024 bytes)
Statically speaking it looks like this...
<table>
<thead>
<tr>
<th>Tools</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Crypto Searcher</td>
<td>Ø</td>
</tr>
<tr>
<td>Draca v0.5.7b</td>
<td>Ø</td>
</tr>
<tr>
<td>Findcrypt v2</td>
<td>Ø</td>
</tr>
<tr>
<td>Hash &amp; Crypto Detector v1.4</td>
<td>Ø</td>
</tr>
<tr>
<td>PEiD KANAL v2.92</td>
<td>Ø</td>
</tr>
<tr>
<td>Kerckhoffs</td>
<td>Ø</td>
</tr>
<tr>
<td>Signsrch 0.1.7</td>
<td>Ø</td>
</tr>
<tr>
<td>SnD Crypto Scanner v0.5b</td>
<td>Ø</td>
</tr>
<tr>
<td>Tools</td>
<td>Answer</td>
</tr>
<tr>
<td>--------------------------------------------</td>
<td>--------</td>
</tr>
<tr>
<td>Crypto Searcher</td>
<td>Ø</td>
</tr>
<tr>
<td>Draca v0.5.7b</td>
<td>Ø</td>
</tr>
<tr>
<td>Findcrypt v2</td>
<td>Ø</td>
</tr>
<tr>
<td>Hash &amp; Crypto Detector v1.4</td>
<td>Ø</td>
</tr>
<tr>
<td>PEiD KANAL v2.92</td>
<td>Ø</td>
</tr>
<tr>
<td>Kerckhoffs</td>
<td>Ø</td>
</tr>
<tr>
<td>Signsrch 0.1.7</td>
<td>Ø</td>
</tr>
<tr>
<td>SnD Crypto Scanner v0.5b</td>
<td>Ø</td>
</tr>
</tbody>
</table>

Let’s try our tool...
• We extracted from our toy program:
• We extracted from our toy program:
• We extracted from our toy program:
• We extracted from our toy program:
STARTED AT
2012-04-08 19:34:12.959000
** Crypto Algorithm Identification starting !**
3 input parameters
3 output parameters
All possible input values generation... Done!
All possible output values generation... Done!
Build internal structure... Done!
Comparison phase starting... Test for RC4...

** Found RC4 **

==> Key (19 bytes) : 53757065724b657949734153757065724b6579

==> Crypted text (1024 bytes) : 4141414141414141...

==> Decrypted text (1024 bytes) : c1dc63d553c720f6...

ENDED AT
2012-04-08 19:34:12.990000
Sality

• Several internet references about the use of RC4 in the Sality packer.

• Let’s take a look...

(suspense...)

140
Loop 1

Loop 2

Loop 3
<table>
<thead>
<tr>
<th>Address</th>
<th>Opcode</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0046932E</td>
<td>inc</td>
<td>ecx</td>
</tr>
<tr>
<td>0046932E</td>
<td>shl ecx, 1</td>
<td></td>
</tr>
<tr>
<td>0046931E</td>
<td>sub eax, 7B70h</td>
<td></td>
</tr>
<tr>
<td>00469337</td>
<td>and ecx, edi</td>
<td></td>
</tr>
<tr>
<td>00469339</td>
<td>inc ecx</td>
<td></td>
</tr>
<tr>
<td>0046933A</td>
<td>shld ecx, eax, cl</td>
<td></td>
</tr>
<tr>
<td>0046933D</td>
<td>sub eax, 6151h</td>
<td></td>
</tr>
<tr>
<td>00469343</td>
<td>inc ecx</td>
<td></td>
</tr>
<tr>
<td>00469344</td>
<td>and ecx, edi</td>
<td></td>
</tr>
<tr>
<td>00469346</td>
<td>test eax, 26DF4C7Dh</td>
<td></td>
</tr>
<tr>
<td>0046934C</td>
<td>sub eax, 0E3C8h</td>
<td></td>
</tr>
<tr>
<td>00469352</td>
<td>and eax, 0BE572435h</td>
<td></td>
</tr>
<tr>
<td>00469358</td>
<td>bsf eax, eax</td>
<td></td>
</tr>
<tr>
<td>0046935E</td>
<td>shl eax, 60h</td>
<td></td>
</tr>
<tr>
<td>00469364</td>
<td>and eax, edi</td>
<td></td>
</tr>
<tr>
<td>00469366</td>
<td>bts eax, 15h</td>
<td></td>
</tr>
<tr>
<td>0046936C</td>
<td>bts eax, 0CDh</td>
<td></td>
</tr>
<tr>
<td>0046937E</td>
<td>sub eax, 0F816h</td>
<td></td>
</tr>
<tr>
<td>00469374</td>
<td>and eax, edi</td>
<td></td>
</tr>
<tr>
<td>00469376</td>
<td>jmp short loc_469379</td>
<td></td>
</tr>
</tbody>
</table>

**Hmpf.. Let’s try!**
TRACER

Execution Trace

CRYPTO EXTRATION

Unknown Algorithm

(Multi-loops)
For the previous 3 loops, we extracted one algorithm:
For the previous 3 loops, we extracted one algorithm:
For the previous 3 loops, we extracted one algorithm:
For the previous 3 loops, we extracted one algorithm:

X86 ExecutableCode!
For the previous 3 loops, we extracted one algorithm:

X86 ExecutableCode!
For the previous 3 loops, we extracted one algorithm:

X86 ExecutableCode!
Unknown Algorithm → IDENTIFICATION → ...
STARTED AT
2012-04-10 16:29:05.135000

** Crypto Algorithm Identification starting !**
5 input parameters
4 output parameters
All possible input values generation... Done!
All possible output values generation... Done!
Build internal structure... Done!
Comparison phase starting... Test for RC4...

** Found RC4 **

```plaintext
>>> Key (8 bytes) : b8a2baa789850cea

>>> Crypted text (57066 bytes) : d2276d92e4cb5446...

>>> Decrypted text (57066 bytes) : e80000000005d81ed...
```

ENDED AT
2012-04-10 16:29:06.929000
RC4 extracted from two Salty binaries
RC4 extracted from two Sality binaries
RC4 extracted from two Sality binaries
RC4 extracted from two Sality binaries
RC4 extracted from two Sality binaries
RC4 extracted from two Sality binaries

Unknown Algorithm (3 Loops)

Crypto parameters always at the same offsets!
EXAMPLE WHERE IT DOESN’T COULD WORK
Compression Algorithms

- Compression algorithms have also a very particular input-output relationship.

- That should work too!
Fail... Why?
Fail... Why?
Fail... Why?
Method Recap

1. We collect an execution trace.

2. We extract possible cryptographic algorithms with their parameter values.

3. We compare the input-output relationship with known algorithms.

We prove that the program behaves like a known crypto algorithm during one particular execution path.
Conclusion (1)

• Interesting alternative to syntactic-identification for crypto algorithms:
  – Resistance against usual obfuscation techniques.
  – Gives the exact parameters.

• Pure dynamic technique, you have to know how to exhibit interesting execution paths.

• It is easy to bypass, like any program analysis technique 😞
Conclusion (2)

• The identification process itself is generic:
  – Collect the execution trace
  – Extract the type of code you are looking for (here is the magic)
  – Get I/O values
  – Compare with reference implementations


  http://code.google.com/p/kerckhoffs/
What’s Next ?

• That’s only the beginning! Just wanted to show that it is feasible and useful.

• Extension to other crypto algorithms.

• Compression algorithms ? What should be the correct criteria to extract compression code ?
  (my 2 cts: natural loops, that is back-edges on CFG)

• If we use several criteria to extract interesting code, how to combine them ?

• How to use the analyst knowledge ?

• Make a real tool. This one is just a PoC.
Thank you for your attention ;-)
### Performances

<table>
<thead>
<tr>
<th></th>
<th>Sality 1</th>
<th>Sality 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trace Size (instructions)</td>
<td>~1M</td>
<td>~4M</td>
</tr>
<tr>
<td>Time To Trace</td>
<td>5mn</td>
<td>10mn</td>
</tr>
<tr>
<td>Time To Extract Crypto Algorithm</td>
<td>4h</td>
<td>15h</td>
</tr>
<tr>
<td>Time To Identify</td>
<td>3mn</td>
<td>4mn</td>
</tr>
</tbody>
</table>

- The tool is just a PoC, no optimization **at all**.
- When the analysts knows where the algorithm is, it will reduce the trace size.
Answer: Brute-Force

• We generate any possible input values by appending together \(A\) inputs, it gives a set \(\text{IN}\).

• Same thing for output values, in the set \(\text{OUT}\).

• For each reference implementation \(F\):
  – We select the \(\text{IN}\) values that can fit into \(F\) parameters.
  – We execute \(F\) on each possible input value combination.
  – If \(F\) output is something in \(\text{OUT}\), it is a success.
For example:

- For each known cryptographic algorithm A:
For example:

- For each known cryptographic algorithm A:
  - We execute a reference implementation of A on 0x42 and 0xCAFEBABE.
For example:

- For each known cryptographic algorithm $A$:
  - We execute a reference implementation of $A$ on 0x42 and 0xCAFEBAEBE.
  - We check if the output is 0xDEADBEEF.
For example:

- For each known cryptographic algorithm $A$:
  - We execute a reference implementation of $A$ on $0x42$ and $0xCAFEBABE$.
  - We check if the output is $0xDEADBEEF$.
  - If so, **we have proved that $P$ implements $A$ on these particular input values.**