Cryptographic Function Identification in Obfuscated Binary Programs

REcon 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

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
Vous 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 = 1001111000110111011100110111001 = 9e3779b9. P64 = 10110111110000101000101100001010001011101101001010110110111...
Vous avez consulté cette page le 05/04/12.

books.google.fr/books?id=0470852852...
Man Young Rhee - 2003 - Computers - 405 pages

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> ...

TEA Encryption Algorithm, Source code
www.shokhirev.com/nikolai/.../uTeaSet_pas.html - Traduire cette page
... XTeaEncrypt/XTeaDecrypt Thanks to Pedro Gimeno Fortea <parigalo@formauri.es>
} interface const Delta: longword = $9e3779b9; type TLong2 = array[0.
<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?
No particular constants

What about this one?
<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?

(If so, we could use it in obfuscated programs!)
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 deterministic 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.

- First idea: execute $P$ on all possible input states and check if the outputs are the same than a known cryptographic algorithm.

  (not realistic!)
Let’s say \( P \) is a program implementing an unknown cryptographic algorithm.

First idea: execute \( P \) on all possible input states and check if the outputs are the same than a known cryptographic algorithm.

\((not\ realistic!)\)

But we can observe one particular \( P \) execution and collect its input-output parameter values...
For example:
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 $P$?
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 P?

- 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 0012bd98</td>
<td>esp 0012bd94</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>12bd94</td>
<td>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></td>
</tr>
<tr>
<td></td>
<td></td>
<td>esp 0012bd84</td>
<td></td>
<td>12bd80</td>
<td>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.
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)

• 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?

Win32.Mebroot
Loops?

Win32.Mebroot

Unrolling optimization

```
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
```
Loops?

Win32.Mebroot

Unrolling optimization

```
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.)
### Example

#### Execution Trace

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

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>

Iteration 1

Iteration 2
### Example

#### Execution Trace

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</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>

**Loop**

- **Iteration 1**
  - 401325: add ebx, edi
  - 401327: sub edx, ebx
  - 401329: dec dword ptr [ebp+0xc]
  - 40132c: jnz 0x401325

- **Iteration 2**
  - 401325: add ebx, edi
  - 401327: sub edx, ebx
  - 401329: dec dword ptr [ebp+0xc]
  - 40132c: jnz 0x401325
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
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
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.
   
   \textit{(non trivial, language w.w is non-context-free)}

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 \((\text{too large!})\)
  2. And if they are manipulated by the 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 the same instruction in the loop body.*

```
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 the same instruction in the loop body.

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

Iteration 1

Iteration 2
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 the same instruction in the loop body.

```
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 the same instruction in the loop body.*

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

add ebx, edi
mov eax, [ebx]
...
```
Loop Parameters (4)

• A parameter is then defined by:
  
  – An identifier: “(memory address | register name):size”

  – A value
Let’s Recap With a Use-Case

• Tiny Encryption Algorithm:
  – Block cipher
  – 16-byte key
  – 8-byte input text
  – Magic constant *delta* (0x9E3779B9)

• We built a toy program calling the TEA decryption function on:
  – Key: `0xDEADBEE1...DEADBEE4`
  – Encrypted text: `0x0123456789ABCDEF`
Step 1 : Gather Execution Trace

First instruction

Last instruction
Step 2 : Recognize Loops

Machine instruction sequence B is repeated
Step 2: Recognize Loops
Step 3 : Define Loop Parameters

Each loop is then a possible cryptographic algorithm!
Final Model
Final Model
Final Model
STEP 3: CRYPTO ALGORITHM IDENTIFICATION
**Input 1:** unknown algorithm A with its parameter values
**Input 1:** unknown algorithm A with its parameter values

```
Input: 0x12ff44:4 0x12ff48:4 0x12ff64:4 0x12ff68:4 0x12ff40:4 0x12ff5c:4 0x12ff60:4 0x12ff38:4 ebp:4
  00000020 01234567 deadbee3 deadbee4 89abcdef deadbee1 deadbee2 9e3779b9 0012ff4c
```

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

```
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 combination of A output values?
Question

• Is there a way to combine \( A \) input values such that \( \text{tea}() \), \( \text{xtea}() \) or \( \text{rc4}() \) would produce a combination of \( A \) 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 $tea()$, $xtea()$ or $rc4()$ would produce a combination of 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 \textit{tea()}, \textit{xtea()} or \textit{rc4()} would produce a combination of 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, 00000020deadbee3,..
   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, 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,...*
1. **Generate all possible values with A input parameters:**
   1. Length 4: \texttt{00000020, 01234567, deadbee3...}
   2. Length 8: \texttt{0000002001234567, 00000020deadbee3,...}
   3. ...

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

3. **For TEA reference implementation:**
   1. Possible input texts (8 bytes): \texttt{00000002001234567,...}
   2. Possible keys (16 bytes): ...
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): ...
   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: 0000020, 01234567, deadbee3...
   2. Length 8: 000002001234567, 00000200deadbee3,...
   3. ...

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

3. **For TEA reference implementation:**
   1. Possible input texts (8 bytes): 000002001234567,...
   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 **

```plaintext
  ===> 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
• 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!)
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]) ^ (y+sum) ^ ((y>>5)+k[3])
```

Storm Worm implementation

```
z== 16 * y + (y ^ *(key + 8)) + (sum ^ (y >> 5)) + *(key + 12)
```
Original TEA source code

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

Storm Worm implementation

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

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

Storm Worm implementation

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

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

Storm Worm implementation

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

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 ?)
Classic TEA operations

TEA round number

TEA classic constant
(delta * round number)

= sub [ebp+arg_0], 0x9E3779B9
Classic TEA operations

Let’s try our tool...

TEA classic constant
(delta * round number)

TEA round number

$= \text{sub} \ [\text{ebp+arg}_0], 0x9E3779B9$
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!)
STARTED AT
2012-04-08 16:05:14.288000
※※ 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... Unknown Algorithm stays unknown!
Done!

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!
Tiny Encryption Algorithm

From Wikipedia, the free encyclopedia

In cryptography, the Tiny Encryption Algorithm (TEA) is a very simple block cipher composed of just 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 16-bit integers in ur出租 [9]) and a suggested 64 rounds, typically implemented in pairs exactly the same way for each cycle. Different multiple implementations exist. The magic constant, 2654435769 or 9E3779B9, was chosen to maximize the number of rounds, and reduces the effective key size to only 126 bits. As a result, TEA is now considered broken for use in space invader hacking Microsoft’s Xbox game console, where the cipher requires \(2^{23}\) chosen plaintexts under a related-key pair 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»[1]
- 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-ветераном, который уверенными шагами вышел на свет и прикоснулся к истории российской хак-сцены....

ТЕА: блочный шифр своими руками

Дата: 22.04.2004

В данном тексте хотелось бы затронуть такую животрепещущую 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 i
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 ; Делимся
pop edi ; Вынимаем i
mov v0,ebx ; Кладём
mov v1,ecx ; В отведённом
ret ; Возвращаем
push edi; Сохраняем
mov ebx, v0; Кладём
mov ecx, v1; В ecx кл.
mov edx, 9e3779b9h; Кладём
mov eax, edx; Кладём
shr 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; Делимся
pop edi; Вынимаем из
mov v0, ebx; Кладём
mov v1, ecx; В отведён
ret; Возвращаем

Storm Worm

loc_1FA1:
mov ebp, ebx
shr 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, ebp
sub ecx, [esi + 0Ch]
mov ebp, ecx
shr ebp, 4
sub ebx, ebp
mov ebp, [esi]
xor ebp, ecx
sub ebx, ebp
mov ebp, ecx
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
retn
MORE EXAMPLES!

RC4
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...
Step 1: Gather Execution Trace
Step 2 : Recognize Loops
Step 2 : Recognize Loops
Step 3 : Define Loop Parameters
Step 4: Connect Loops With Data-Flow
Loop Data Flow Graph (oriented, acyclic)
We consider each path in the graph as a possible cryptographic algorithm!

(in order to deal with algorithm combinations)
We consider each path in the graph as a possible cryptographic algorithm!

(in order to deal with algorithm combinations)
We consider each path in the graph as a possible cryptographic algorithm!

(in order to deal with algorithm combinations)
We consider each path in the graph as a possible cryptographic algorithm!

(in order to deal with algorithm combinations)
We consider each path in the graph as a possible cryptographic algorithm!

(in order to deal with algorithm combinations)
Final model for the longest path
Final model for the longest path
Final model for the longest path

Key

Input text

Output text
Final model for the longest path
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
Win32.Sality.AA

• Several internet references about the use of RC4 in Sality.AA protection layers.

• Let’s take a look...

(suspense...)
Loop 1

Loop 2

Loop 3
Hmpf.. Let’s try!
Sality Sample → TRACER → Execution Trace → CRYPTO EXTRACTION → Unknown Algorithms (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:
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 **

===>< 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 Sality.AA binaries
RC4 extracted from two Sality.AA binaries
RC4 extracted from two Sality.AA binaries
RC4 extracted from two Sality.AA binaries

Unknown Algorithm (3 Loops)
RC4 extracted from two Sality.AA binaries

Crypto parameters always at the same offsets!
MORE EXAMPLES!
Remember This?

```
0x12ff44:4 00000020
0x12ff48:4 01234567
0x12ff64:4 deadbee3
deadbee4
deadbee1
deadbee2

eax:4 6e089494
ecx:4 df5ec153
edx:4 00000000
0x12ff4c:4 0012ff4c
```

Unknown Algorithm
The magic TEA constant (*delta*) and the round number are seen as input parameters, because they are initialized *before* the loop and used inside.
Modified TEA Implementation

• \( \delta = 0x12345678 \) \textit{(normally} \( 0x9E3779B9 \))
• \( \text{round number} = 16 \) \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):
...
```
Example: Mozilla CTF

• Challenge “Awesome Corp. Secured Ranges”

• Binary program protected by PE Spin

• In the core binary, a strange algorithm...
LOOP 1

mov [esp+240h+var_22C], 7

loc_401649:
    mov  ebx, [esp+240h+var_228]
    mov  eax, esi
    shr  eax, 5
    mov  edi, esi
    shl  edi, 4
    xor  eax, edi
    mov  edi, [esp+240h+pcbBuffer]
    shr  edi, 0Bh
    and  edi, 3
    mov  edi, [ebx+edi*4]
    add  edi, [esp+240h+pcbBuffer]
    sub  [esp+240h+pcbBuffer], 61C88646h
    add  eax, esi
    xor  eax, edi
    sub  edx, eax
    mov  eax, edx
    shr  eax, 5
    mov  edi, edx
    shl  edi, 4
    xor  eax, edi
    mov  edi, [esp+240h+pcbBuffer]
    and  edi, 3
    mov  edi, [ebx+edi*4]
    add  edi, [esp+240h+pcbBuffer]
    add  eax, edx
    xor  eax, edi
    sub  esi, eax
    dec  [esp+240h+var_22C]
    jnz  short loc_401649
Common TEA operations
LOOP 1

Common TEA operations

Not the TEA round number

Not the TEA delta
LOOP 2

TEA round number!
Loop Data Flow Graph

3 possible cryptographic algorithms
STARTED AT
2012-06-01 17:55:57.729000

*** Crypto Algorithm Identification starting !*** 6 inputs - 5 outputs
Comparison phase starting... Test for XTEA *modified* decryption... Unknown Algorithm stays unknown!

*** Crypto Algorithm Identification starting !*** 6 inputs - 5 outputs
Comparison phase starting... Test for XTEA *modified* decryption...
*** Found XTEA *extended* decryption ***

==> Key (16 bytes) : 0x09c195dbb17945f5503a3b2bc61fa033

==> Delta (4 bytes) : 0x61c88646

==> Round number (4 bytes) : 0x00000007

==> Crypted text (8 bytes) : 0x2adc69d7832172aa

==> Decrypted text (8 bytes) : 0x75d081d699f152ae

*** Crypto Algorithm Identification starting !*** 6 inputs - 5 outputs
Comparison phase starting... Test for XTEA *modified* decryption...
*** Found XTEA *extended* decryption ***

==> Key (16 bytes) : 0x09c195dbb17945f5503a3b2bc61fa033

==> Delta (4 bytes) : 0x61c88646

==> Round number (4 bytes) : 0x00000020

==> Crypted text (8 bytes) : 0x75d081d699f152ae

==> Decrypted text (8 bytes) : 0x911b822660cfe1f1

ENDED AT
2012-06-01 17:57:43.659000
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 a program behaves like a known crypto algorithm during one particular execution path.
Conclusion (1)

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

• As any 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 ? (1)

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

• What about more complex algorithms ? What about hash functions ? Compression algorithms ?

• What about proprietary algorithms ?
What’s Next? (2)

• Make a real tool. This one is just a PoC.

• How to use the analyst knowledge? In practice he often knows where the crypto is, analyzing a complete execution trace is more an academic hobby.
Thank you for your attention ;-) 

j04n.calvet@gmail.com