CRYPTON – EXPOSING MALWARE'S DEEPEST SECRETS

Julia Karpin & Anna Dorfman
F5 Networks, Israel

Email {julia.karpin, adasaster}@gmail.com

ABSTRACT

As malware researchers, a significant part of our work is dedicated to reverse engineering various cryptographic algorithms in order to extract malware's encrypted content.

Revealing this content provides access to the heart of the malware: strings, *Windows* API calls, DGA algorithms, communication protocols, and in financial malware, lists of targeted institutions and man-in-the-browser injections.

Sophisticated malware authors know that we're after this data, which is why they invest considerable effort into constantly changing their encryption routines and designing customized encryption implementation of cryptographic algorithms. Even the smallest change sometimes requires significant work on the part of the malware researcher: reverse engineering must be applied to reconstruct the encryption scheme.

Over the years, plug-ins and tools were developed to aid with solving this issue. Some were highly academic endeavours that relied on complicated algorithms to identify cryptography, but they were not well adapted for real-world usage; others relied on signature checks to locate specific algorithms. Our motivation was to find a lightweight and practical implementation that would effectively speed up the research process.

That's why we developed an automated approach, based on a heuristic method of detecting such cryptographic algorithms regardless of the type of algorithm used to extract their plain text output.

The implementation of this approach saves a lot of valuable research time by letting the malware decrypt the data for us!

Our implementation, 'Crypton', works by first unpacking the malware, then following injected code and memory allocations to identify blocks of cryptographic code, and finally inspecting the allocations for decrypted data.

Our tool will follow all the processes created and injected by the malware as the decryption routine may take place in any one of them – therefore we must follow any execution flow.

In this paper, we will describe the concept and the architecture of the Crypton tool and our IDAPython script that identifies all crypto blocks inside a memory dump.

INTRODUCTION

Ever since online banking became the dominant way to make financial transactions, fraudsters have been on the lookout for innovative ways to hijack those transactions, steal users' credentials and, most importantly, empty their bank accounts.

Both the online banking industry and security experts have been coming up with different solutions to this problem. This created an arms race that is still ongoing.

For instance: the fraudsters planted keyloggers on users' machines to intercept key strokes, log sensitive user data and send it back to the command and control (C&C) server of the malware.

As a countermeasure, the banks added virtual keyboards on their sites so that the key strokes wouldn't come from the keyboard.

The introduction of SSL posed a significant challenge for fraudsters as it prevented them from conducting classic MITM attacks. To overcome this obstacle, they incorporated 'man-in-the-browser' attacks – function hooks in the browser, intercepting outgoing traffic before it was encrypted and incoming traffic after it was decrypted. This gave them the ability to observe all the passing data, and inject additional malicious JavaScript code into the bank's page once it was loaded by the browser.

In many cases, the malicious JavaScript code is responsible for modifying the page to trick the user into giving away additional information the bank usually doesn't ask for, as well as hiding security items. As web technologies continue to evolve, acquiring new features, so does the skillset of the malware developers. Hence, it is crucial to stay up to date with those web technologies as well as their abuse by fraudsters. Nowadays, fraudsters use advanced techniques such as web-rootkits [1], self-removing scripts [2], redirects [3], etc.

The content of these JavaScript configurations became the heart of the fraud scheme – and therefore hiding them became a major part of this arms race.

As hiding the injected JavaScript from the web page's source is currently impossible, the fraudsters resort to the methods that are available: developing stronger obfuscation techniques and encrypting the JavaScript configuration that is downloaded from the C&C.

```
set_url https://www.[redacted].pl/[redacted]-web/* GP
data_before
<head>
data_end
data_inject
<script id="myqwe1">
window.rem777bname = '%BOTUID%';
window.rem777ddeell = function (a){document.getElementById(a).parentNode.removeChild(document.getElementById(a));};
</script>
<script id="myqwe4" src="https://ajax.googleapis.com/ajax/libs/jquery/2.1.1/jquery.min.js"></script>
<script id="myqwe2" src="https://[redacted].ru/pl/cen_frr.js"></script>
<script id="myqwe3">
delete $;delete jQuery;
window.rem777ddeell("myqwe1");window.rem777ddeell("myqwe2");window.rem777ddeell("myqwe4");window.rem777ddeell("myqwe3");
delete rem777bname;delete rem777ddeell;
</script>
data_end
```

Figure 1: A snippet from Tinba's webinjects.

ENCRYPT LIKE EVERYONE IS WATCHING

All major malware families employ encryption to make the task of reverse engineering them more cumbersome.

The encryption may be applied to strings, *Windows* APIs and webinject configurations, and in most cases, decrypting them is a crucial part of understanding the code flow. The most common method of dealing with this is a combination of live malware debugging and reconstructing the code from the disassembly view.

Once all the stages of encryption and/or compression are known, it is possible to automate the process of extracting the plaintext information. This is done by writing scripts that implement all the stages, accept the encrypted data and possible keys/IVs as parameters and output the plaintext data.

While this may sound easy, here's how it gets complicated:

1. Each malware family stores the encrypted data in a different place – it could be a registry key, a system file or in memory.
2. Each malware family employs various stages and different encryption algorithms.
3. Some malware families employ proprietary encryption schemes which literally need to be reconstructed from scratch as known implementations don't exist.
4. The moment in runtime when the encryption key/IV becomes visible differs among malware families.
5. Some malware families distribute the decryption stages between processes. In some cases, finding the code that contains the encryption scheme can be much more time consuming than reverse engineering the algorithm itself.

All of the above changes frequently, per malware variant.

As a result, a custom script is implemented for each malware type and in some case, even for specific variants. Thus, the error handling process and bug fixing is distinct for each script, requiring separate version control.

Every time a slight change is inserted into the malware's encryption scheme, the previous script implementation becomes obsolete and new research must be conducted.

After applying this process repeatedly, we started considering more generic approaches.

First, we started looking for the known applications of any approaches that are not dependent on the algorithm.

Secondly, we searched for an approach that wouldn't be dependent on the malware type or on the place and point in time where the decryption takes place.

In the following section, we will present the overview of the existing tools and approaches.

RELATED WORK

In our efforts to improve and speed up the process of extracting encrypted data, we looked for the appropriate solutions among existing tools both for static and dynamic analysis. Basically, identifying cryptographic code blocks relies on the fact that cryptographic implementations share common characteristics. Even though most malware types use strong obfuscation techniques, the presence of one or more of these features cannot be completely hidden inside the code:

- Usage of known *Windows* APIs for crypto (e.g. in Public Key cryptography [4]).
- Applying operations on known constant numbers.
- Lots of arithmetic operations spread over relatively short function space.
- Well known crypto implementation parts compiled inside the malicious code.
- Characteristics of the encrypted data buffers (e.g. entropy) prior to the execution or during various stages of the execution of crypto functions.

The academic approaches that we've found were focused mainly on improving the accuracy of identifying specific algorithms. This is achieved by finding diverse ways of approximating the known algorithm implementations to the inspected code artifacts, or by dynamically inspecting the entropy of buffers [5]. Although these approaches have potential, implementations are computationally demanding and unreliable, and as such, these methods are not a viable solution for the malware research industry.

On the other hand, commonly used lightweight plug-ins for static and dynamic analysis are based mainly on signatures, signing constants and exact code parts. Highly accurate signatures can be created for encryption and compression algorithms like CRC32, zlib, SHA family and exact defined S-boxes usage (AES, DES, Skipjack, etc.) [6, 7].

These tools provide an efficient and quick way to find notable crypto usage in executables. But malware developers' awareness of these tools leads them to invest effort into rapidly introducing changes to their encryption schemes. Because of the signature-based nature of these approaches, even a slightly modified crypto implementation inside the malicious code may not be identified.

SUGGESTED SOLUTION

Our goal was to find an approach which would be easy to integrate into the malware research process. The following is the description of the flow in our lab:

- Obtaining samples.
- Classifying them according to families, such as: Zeus, Dridex, Ramnit, etc.
- Configuring a lab environment according to the malware family. Some malware will only execute on machines with specific files, registry values, installed software, etc.
- Executing the malware with a custom script that has been made to handle the particular family. The output of the script is the plaintext webinjects configuration. The script will either connect to infected processes as a debugger or handle the encrypted configuration data, obtaining it from the place where it is stored.

We would like to have a single script that can handle all malware types and which has the same launching point for all malware families. At the very least we would like to minimize the research time per malware variant.

IT'S ALL ABOUT MATH (IN A LOOP)

If our goal is to accommodate all cryptographic algorithms, the identification of specific algorithms is redundant. For our

purpose, it is good enough to acknowledge the existence of any such algorithm and follow its code flow to extract its output.

What do all cryptographic algorithms have in common?

```
case 12:

    for( i = 0; i < 8; i++, RK += 6 )
    {
        RK[6]  = RK[0] ^ RCON[i] ^
        ( (uint32_t) FSb[ ( RK[5] >>  8 ) & 0xFF ]       ) ^
        ( (uint32_t) FSb[ ( RK[5] >> 16 ) & 0xFF ] <<  8 ) ^
        ( (uint32_t) FSb[ ( RK[5] >> 24 ) & 0xFF ] << 16 ) ^
        ( (uint32_t) FSb[ ( RK[5]       ) & 0xFF ] << 24 );

        RK[7]  = RK[1] ^ RK[6];
        RK[8]  = RK[2] ^ RK[7];
        RK[9]  = RK[3] ^ RK[8];
        RK[10] = RK[4] ^ RK[9];
        RK[11] = RK[5] ^ RK[10];
    }
    break;

case 14:

    for( i = 0; i < 7; i++, RK += 8 )
    {
        RK[8]  = RK[0] ^ RCON[i] ^
        ( (uint32_t) FSb[ ( RK[7] >>  8 ) & 0xFF ]       ) ^
        ( (uint32_t) FSb[ ( RK[7] >> 16 ) & 0xFF ] <<  8 ) ^
        ( (uint32_t) FSb[ ( RK[7] >> 24 ) & 0xFF ] << 16 ) ^
        ( (uint32_t) FSb[ ( RK[7]       ) & 0xFF ] << 24 );

        RK[9]  = RK[1] ^ RK[8];
        RK[10] = RK[2] ^ RK[9];
        RK[11] = RK[3] ^ RK[10];

        RK[12] = RK[4] ^
        ( (uint32_t) FSb[ ( RK[11]       ) & 0xFF ]       ) ^
        ( (uint32_t) FSb[ ( RK[11] >>  8 ) & 0xFF ] <<  8 ) ^
        ( (uint32_t) FSb[ ( RK[11] >> 16 ) & 0xFF ] << 16 ) ^
        ( (uint32_t) FSb[ ( RK[11] >> 24 ) & 0xFF ] << 24 );

        RK[13] = RK[5] ^ RK[12];
        RK[14] = RK[6] ^ RK[13];
        RK[15] = RK[7] ^ RK[14];
    }
```

Figure 2: The C implementation of AES(Rijandel).

```
// Key Scheduling Algorithm
// Input: state - the state used to generate the keystream
//        key - Key to use to initialize the state
//        len - length of key in bytes
void ksa(unsigned char state[], unsigned char key[], int len)
{
   int i,j=0,t;

   for (i=0; i < 256; ++i)
      state[i] = i;
   for (i=0; i < 256; ++i) {
      j = (j + state[i] + key[i % len]) % 256;
      t = state[i];
      state[i] = state[j];
      state[j] = t;
   }
}

// Pseudo-Random Generator Algorithm
// Input: state - the state used to generate the keystream
//        out - Must be of at least "len" length
//        len - number of bytes to generate
void prga(unsigned char state[], unsigned char out[], int len)
{
   int i=0,j=0,x,t;
   unsigned char key;

   for (x=0; x < len; ++x)  {
      i = (i + 1) % 256;
      j = (j + state[i]) % 256;
      t = state[i];
      state[i] = state[j];
      state[j] = t;
      out[x] = state[(state[i] + state[j]) % 256];
   }
}
```

Figure 3: The C implementation of RC4.

Let's examine the disassembled code for the two algorithms shown in Figures 2 and 3 – see Figure 4 on the next page.

While in high-level languages the algorithms may seem different, when looking at assembly instructions we can narrow them down to mathematical operations, mostly XORing but also shifting, multiplying and so on. These operations are applied on the bytes of the encrypted buffer in a loop. This is what differentiates regular program code from cryptography-related code and it has been widely discussed in research papers [8, 9].

We used this knowledge to build a static analysis tool (an IDAPython script) that can identify, by finding crypto code blocks, interesting starting points for static analysis. We also used it to build a dynamic tool which we integrated into our automatic configuration extraction process.

The implementation specifics of the tools will be discussed in the section 'Crypton to the rescue'.

BUT WHERE SHOULD WE START?

Now that we can recognize crypto-related code, we need to decide on where and when to apply this. As those operations are applied to buffers, we considered them as the focus of inspection.

Several approaches were considered for this end:

1. Monitoring Windows APIs

Since malware configurations appear in several known locations, e.g. the file system and network communication, it is possible to narrow down the *Windows* APIs to a subset of relevant I/O operations. This can be achieved by monitoring functions such as: CreateFile, RegSetValue, etc. and following the buffers involved. If the buffer originated from a relevant API and later on was modified by crypto-related code, this buffer is a candidate for being the desired output.

Even though this approach was implemented in previous research [10], we decided to abandon it as it is less practical for our needs. After listing the relevant *Windows* APIs, we realized that the subset requires constant updating as *Windows* OS keeps updating DLLs, browsers update their own relevant APIs, and malware developers are always looking for ways to use less common or even custom APIs.

2. Following buffer allocations

Since our main goal is to find relevant data buffers that have been modified by crypto code, the essential implementation idea is to follow them.

This can be achieved using the following steps:

- Track all the allocations that originated from malicious memory address space, by hooking relevant APIs, e.g. VirtualAlloc.
- Intercept every write operation on these allocations.
- Inspect the code that accessed the buffer to determine if it is crypto logic.
- After the crypto logic code has been executed check if the buffer contains plaintext data.

Using this approach there is no need to traverse all the code.

While trying to implement this approach we encountered the following shortcomings:

- We used page guard on any allocation the malware committed in order to follow buffer write operations.

```
; Attributes: bp-based frame

rc4_xor_box proc near

arg_0= dword ptr  8
arg_4= dword ptr  0Ch
arg_8= dword ptr  10h

push    ebp
mov     ebp, esp
mov     edi, [ebp+arg_0]
mov     esi, [ebp+arg_4]
xor     eax, eax
xor     ebx, ebx
xor     ecx, ecx
xor     edx, edx

loc_6267E:
inc     bl
mov     dl, [edi+ebx]
add     al, dl
mov     cl, [edi+eax]
mov     [edi+ebx], cl
mov     [edi+eax], dl
add     cl, dl
mov     cl, [edi+ecx]
xor     [esi], cl
inc     esi
dec     [ebp+arg_8]
jnz     short loc_6267E

leave
retn    0Ch
rc4_xor_box endp
```

```
movzx   eax, dl
xor     ecx, ds:Rijndael_Te3[eax*4]
mov     eax, esi
xor     ecx, [eax+10h]
mov     eax, edx
shr     eax, 8
mov     [ebp+var_14], ecx
movzx   ecx, al
mov     ecx, ds:Rijndael_Te2[ecx*4]
mov     eax, ebx
shr     eax, 10h
movzx   eax, al
xor     ecx, ds:Rijndael_Te1[eax*4]
mov     eax, [ebp+var_18]
shr     eax, 18h
xor     ecx, ds:Rijndael_Te0[eax*4]
mov     eax, [ebp+var_C]
movzx   eax, al
xor     ecx, ds:Rijndael_Te3[eax*4]
mov     eax, esi
xor     ecx, [eax+14h]
mov     eax, edx
shr     eax, 10h
movzx   eax, al
mov     [ebp+var_10], ecx
mov     ecx, ebx
shr     ecx, 18h
shr     edx, 18h
mov     eax, ds:Rijndael_Te1[eax*4]
xor     eax, ds:Rijndael_Te0[ecx*4]
mov     [ebp+var_4], eax
mov     eax, [ebp+var_C]
mov     ecx, [ebp+var_4]
shr     eax, 8
movzx   eax, al
xor     ecx, ds:Rijndael_Te2[eax*4]
mov     [ebp+var_4], ecx
mov     ecx, [ebp+var_18]
mov     esi, [ebp+var_4]
movzx   eax, cl
shr     ecx, 8
xor     esi, ds:Rijndael_Te3[eax*4]
mov     [ebp+var_4], esi
mov     eax, esi
mov     esi, [ebp+var_8]
xor     eax, [esi+18h]
mov     [ebp+var_4], eax
movzx   eax, cl
mov     ecx, ds:Rijndael_Te0[edx*4]
xor     ecx, ds:Rijndael_Te2[eax*4]
mov     eax, [ebp+var_C]
```

Figure 4: Disassembled code for both algorithms.

Along with the user-mode exceptions we foresaw, this also triggered kernel-level exceptions, which we could not handle from user land. These kernel exceptions caused I/O operations on the buffer to fail.

We considered two approaches for dealing with this issue:

- Disabling all page guards before every syscall (call from user land to kernel land) and enabling them right after the return from kernel land – since most of the I\O operations are not atomic, other threads may intervene and change other buffers while the page guard is off. This will lead to missing important write accesses to the relevant buffers. We tried to suspend all the other threads when entering a syscall and resume them right after, so that when the page guard is off, there are no other running threads. This solution led to synchronization and starvation issues.
- Writing a kernel driver that will handle the flow that leads to a user-mode page fault exception – after considering this option, we learned that when performing an I/O operation, the decision whether to commit the I/O or return an error is made in an internal routine (ProbeForWrite). This routine checks if a user-mode buffer is writable. If we hook this function in order to enable and disable the page guard, we return to the same situation as mentioned above in a multi-threaded environment.

3. Traversing the code prior to execution

Instead of focusing on allocations, we focused on finding the crypto code blocks first and then, during runtime, inspecting the buffer at the beginning and at the end of the code block. If any changes were detected, we inspected the buffer for plaintext. This method was eventually chosen as it works better than the rest.

The following section will focus on the details of the chosen implementation.

CRYPTON FOR THE RESCUE: IMPLEMENTATION DETAILS

Our solution was implemented in Python, using Mario Vilas’ winappdbg framework [11].

1. *Hooking allocation-related APIs* (in order to store all the allocated buffers and heaps)

   Buffers and heap addresses and sizes are only stored if they came from malicious address space.

2. *Static code analysis*

   - Find the start of the malware's address space – this is still in the development stages and is detailed in the next section.
   - Traverse the assembly code in a flat manner and look for loops.
   - For each loop, a crypto score is calculated according to the amount and type of mathematical operation found in its assembly instructions. While calculating the score, it is vital to consider any nested loops and count their score only once.
   - Since the code traversal is flat, regions that don't contain valid code can be treated as code. So, during the loop exploration, we ignore loops that consist of illegal instructions.
   - Breakpoints are inserted at the beginning and at all the return branches of the loop.

3. *Dynamic analysis*

   - The allocation callback stores all the buffers in a list.
   - A hash of each buffer in the buffer list is calculated at the loop start breakpoint and at the loop end breakpoint, these hashes are compared to see if a buffer was changed during the execution of the loop.

4. *ASCII/Unicode check*

   After the modified buffer has been pinpointed, we check the buffer for ASCII/Unicode in order to tell if it is plaintext data.

5. *Process injection flow*

   Crypton is designed to follow the propagation of the malware across the system. Any process creation or remote thread creation will result in attaching a Crypton instance to the newly created process/thread.

DEALING WITH UNPACKING

At the moment, we have specific plug-ins per malware type in order to discover the point in time when the malware first appears unpacked in memory. At this point it is possible to determine the size and location of the malware's address space. Once this information obtained, the static code analysis can commence. The construction of the plug-in requires manual analysis.

For the sake of making this part more generic and automatic, we came up with an algorithm that can deal with unpacking. Generic unpacking is one of the hardest problems of the malware analysis field. Packers are always evolving and employing sophisticated techniques for anti-debugging, polymorphism and hiding the actual malicious code.

To overcome this challenge, we decided to attack this from a unique perspective – rather than looking for unpacked code, we will follow any allocation that contains executable code and apply our Crypton implementation on it. Eventually, it will lead to the unpacked code.

When a new executable region is allocated, it can be either a new address or an overriding of an existing code section. We must recognize the latter and rescan the region accordingly.

CONCLUSION

Although this is our chosen solution, we are aware of its disadvantages, which we carefully considered during our work.

It may be unsuitable for extremely large amounts of code since in that case, the traversal is time consuming. This delay may cause changes in the flow in case of timing checks in the malicious code.

Future improvement will include a more concurrent approach.

We welcome suggestions for improvements, and any feedback.

Crypton was integrated into our virtual environment and was tested for an extended period of time. It proved to be an efficient and practical solution for the financial malware we deal with on a daily basis, including ZeusVm, Trickbot, Ramnit, Atmos and Qakbot.

REFERENCES

[1] https://www.blackhat.com/docs/us-15/materials/us-15-Park-Winning-The-Online-Banking-War-wp.pdf.

[2] https://f5.com/labs/articles/threat-intelligence/malware/webinject-crafting-goes-professional-gozi-sharing-tinba-webinjects-22453.

[3] https://f5.com/labs/articles/threat-intelligence/malware/dridex-botnet-220-campaign-targeting-uk-financials-with-webinjects-22411.

[4] https://en.wikipedia.org/wiki/Public-key_cryptography.

[5] Cryptographic Function Identification in Obfuscated Binary Programs. https://recon.cx/2012/schedule/events/208.en.html.

[6] findcrypt2. https://github.com/vlad902/findcrypt2-with-mmx/blob/master/findcrypt2-with-mmx/findcrypt.cpp.

[7] FindCrypt for OllyDbg. http://www.openrce.org/downloads/details/191/FindCrypt.

[8] IDAScope. https://bitbucket.org/daniel_plohmann/simplifire.idascope.

[9] https://storage.googleapis.com/google-code-archive-downloads/v2/code.google.com/kerckhoffs/Groebert-Automatic.Identification.of.Cryptographic.Primitives.in.Software-27c3-CCC.pdf.

[10] Finding and Extracting Crypto Routines from Malware. https://net.cs.uni-bonn.de/fileadmin/user_upload/leder/ipccc1569262248.pdf.

[11] https://github.com/MarioVilas/winappdbg.