p0sT5n1F3r

Reverse Engineering of a breach

YARIX
a vargroup company

Security Report
10.2019

SUMMARY

YARIX
a vargroup company

1. Introduction

The Yarix team analyzed a very insidious backdoor, recognized today as 100% clean: a complex artifact designed exclusively for the Customer's environment.

The case

During a recent engagement our **Incident Response Team** had the opportunity to analyze a very insidious backdoor implanted in an Apache web server. This allows sniffing HTTPS traffic when it is licitly decrypted by the web-server. What caught our attention was the fact that this component, while revealing many of the features of the Linux/Cdorked.A backdoor, may today be recognized as **100% clean** on Virus Total.

It is not every day that we find ourselves faced with cases like this: we are definitely facing a **complex artifact designed exclusively for the Customer's environment** which, thanks to extensive use of encryption, is not detected by any anti-malware platform, not even the most advanced ones that have conquered the market in recent years with behavioral and / or machine learning algorithms.

Apache backdoors are nothing new: unfortunately, those who are victims of malware such as Linux/Cdorked.A should remember its features and how it interacted with the infamous Blackhole exploit kit. The use of external modules or plugins for web-servers is a known persistence technique, used and abused over the years but still utilized today. Even last year, Palo Alto intelligence sources revealed that the OilRig group used the RGDoor module as a backdoor for the IIS web-server in attacks in the Middle East.

2. First results

The hooks exploited by this module are offered natively by the Apache module.

Static analysis

Let's start from the beginning: what is an Apache module? At high level it can be considered as a sort of library: additional code used to extend native functionalities - in this case of the Web Server.

Today, in the standard package distributions, we find some already installed by default, such as mod_ssl or mod_php. In this specific case we found a module, mod_dir.so, which imitates in all respects the functionality of the standard one but adds others, really insidious.

The framework provided by Apache provides the developer with a series of hooks. These allow to run additional code during the different states of execution of the process.

In particular, the hooks exploited by this module *(img 1)* are offered natively by the Apache framework and therefore we can know what and how they should be used *(ref here)*.

Image 1

Hooks used by the module

```
; Attributes: bp-based frame

InitializationRoutine proc near

var_8           = qword ptr -8

; __unwind {
                push    rbp
                mov     rbp, rsp
                sub     rsp, 10h
                mov     [rbp+var_8], rdi
                mov     ecx, 0Ah
                mov     edx, 0
                mov     esi, 0
                lea     rdi, sub_32ED
                call    _ap_hook_child_init
                mov     ecx, 0Ah
                mov     edx, 0
                mov     esi, 0
                lea     rdi, sub_3A14
                call    _ap_hook_post_config
                mov     ecx, 0Ah
                mov     edx, 0
                mov     esi, 0
                lea     rdi, sub_34AB
                call    _ap_hook_insert_filter
                mov     ecx, 0FFFFFFF6h
                mov     edx, 0
                mov     esi, 0
                lea     rdi, sub_3D17
                call    _ap_hook_handler
                mov     ecx, 0FFFFFFF6h
                mov     edx, 0
                mov     esi, 0
                lea     rdi, sub_47B4
                call    _ap_hook_log_transaction
                mov     ecx, 14h
                mov     edx, 0
                lea     rsi, sub_38C7
                lea     rdi, aP0st5n1f3r ; "p0sT5n1F3r="
                call    _ap_register_input_filter
                mov     ecx, 14h
                mov     edx, 0
                mov     esi, 0
                lea     rdi, sub_316E
                call    _ap_hook_fixups
                leave
                retn
; } // starts at 482F
InitializationRoutine endp
```

• `ap_hook_child_init`: place a hook that executes when a child process is spawned (commonly used for initializing modules after the server has forked).

• `ap_hook_post_config`: place a hook that executes after configuration has been parsed, but before the server has forked.

• `ap_hook_insert_filter`: place a hook that executes when the filter stack is being set up.

• `ap_hook_handler`: place a hook that executes on handling requests.

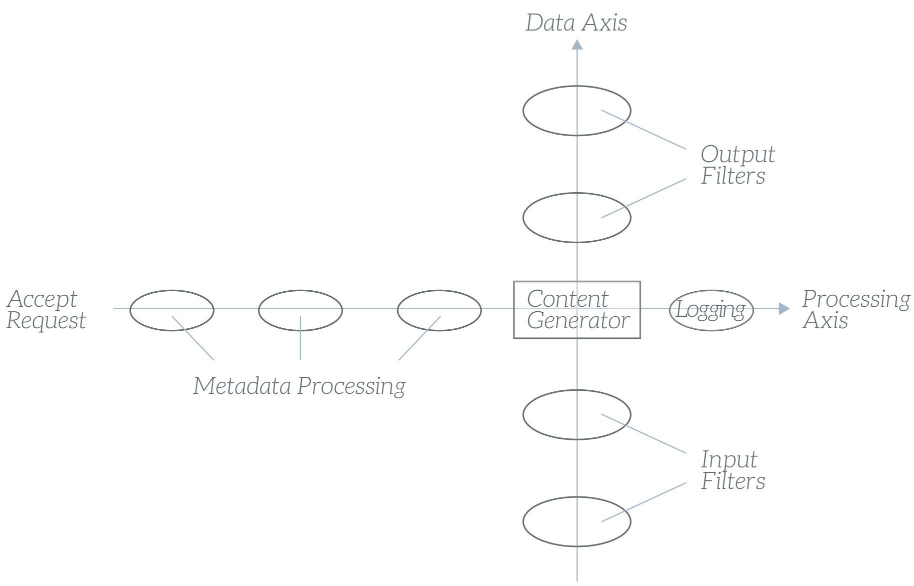
• `ap_hook_log_transaction`: place a hook that executes when the server is about to add a log entry of the current request.

• `ap_hook_register_input_filter`: place a hook that executes a custom function when input is required.

• `ap_hook_fixups`: place a hook that executes right before content generation.

Based on the description of these functions, we have an idea of how the module works *(img 2)*.

Image 2
Request processing in Apache 2
Ref: Apache Tutor



Analyzing the individual functions that are performed by these hooks we can try to understand what this module actually does.

HOOK 1 | ap_hook_child_init

The first function performed is the one that is called by `ap_hook_child_init`. Thanks to IDAPro's capabilities it is possible to understand which functions of the standard library are called.

It is therefore possible to get a high-level idea of the actions performed as soon as Apache creates a child process to handle a request that arrives towards the server port 80 or 443 *(img 3)*.

Image 3
Module Initialization routine

During initialization the module attempts to create a mutex and, if it fails, generates an error which is then logged by the `_ap_log_error` function.
A sort of debug printf on an error_log.

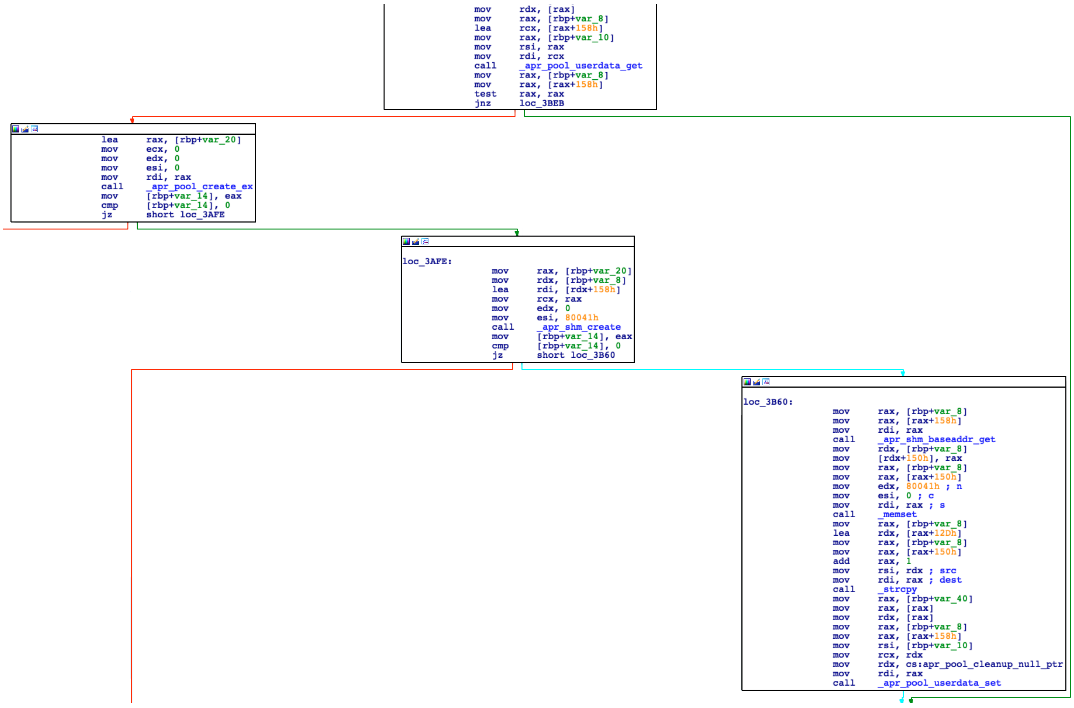
The other function called `apr_shm_baseaddr_get` is more interesting and gets, through the RDI register, the base address of the shared memory segment. Unfortunately, at the moment we are not able to understand much more from static analysis.

HOOK 2 | ap_hook_post_config

The second hook `ap_hook_post_config` that we are going to analyze is executed immediately before the father Apache process reduces its **root** privileges to **www-data**. In fact these privileges serve to allocate portions of memory which will then be used during the operation of the module. It is possible to note the portion of code that allocates a new memory pool *(img 4)*.

Image 4

Portion of code that allocates a memory pool



Until now we found nothing really strange or malicious or interesting. The module begins to show its capabilities in the functions called by the other hooks.

It is here that we meet for the first time the string that from now on will represent the name of our malware: **p0sT5n1F3r=**

Our attention is immediately struck by this string which is passed as an argument to the function `ap_register_input_filter`. The official documentation shows that the prototype of this function is:

```
ap_register_input_filter
("filter name", filter_function, AP_FTYPE_CONTENT)
```

The first parameter is the name of the filter, the second is the function performed by the filter and the third is the type of filter. So we know that:

- the module registers an input filter
- the filter is called **p0sT5n1F3r=**
- the filter, when invoked, executes the function **sub_38C7**
- the filter acts on the **content** of the request and not on its headers

Before going into the details of how it works let's try to get an overview of the other functions used by the module, so as to focus on what is really interesting. Reverse engineering in general is an activity that requires a lot, a lot of time and the risk of getting lost in the technicalities of assembly language is very high.

HOOK 3 | ap_hook_insert_filter

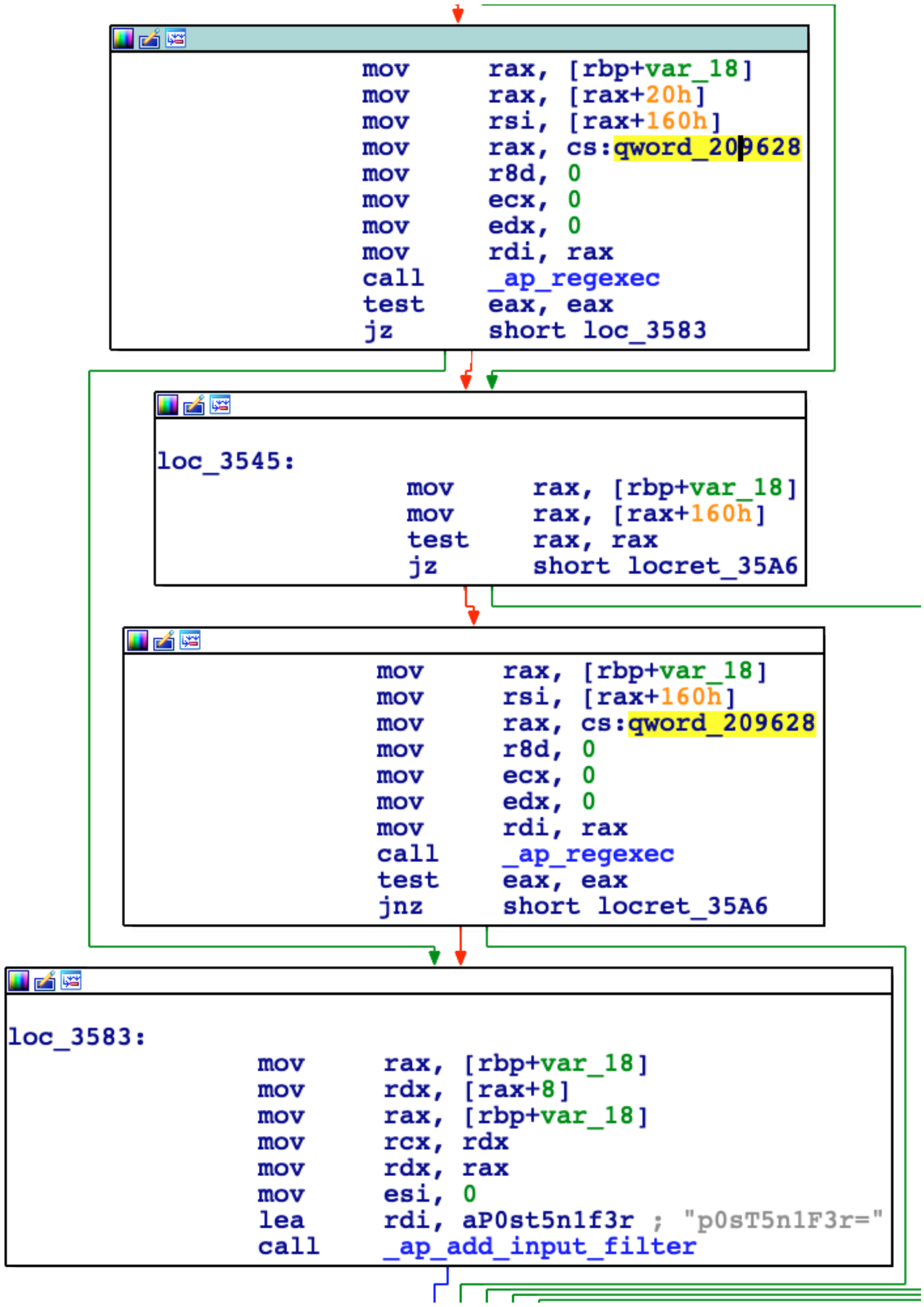
Let's take a look at the function `ap_hook_insert_filter` now. Its prototype is:

```
ap_hook_insert_filter(filter_insert, NULL, NULL, APR_HOOK_MIDDLE)
```

Thus, the inserted filter is given by the function passed as the first argument, ie the function **sub_34AB**. This function takes care of building the filter which will then be activated inside the module and in fact the function `ap_regexec` is used, whose role is that of "**Match to NUL-terminated string against a pre-compiled regex**" *(img 5)*.

Image 5

Role of the function ap_regexec



It is clear that the regular expression passed to the function resides in the position of the **qword_209628** but it is clearly obtained at runtime, since in static analysis this location is empty.

What did we understand until now?
From the static analysis of these functions we were able to understand that the module:

- is characterized by the string p0sT5n1F3r=
- takes care of inserting an input filter within the task of processing requests coming to the web-server
- acts on the body of requests and not on headers
- the filter is activated only if it meets the exact match of a string that is obtained at runtime.

3. An important discovery

Continuing analysis

The approach we are following was useful to begin to understand how the module works because the code was not obfuscated and none of the functions, custom or standard libraries, were resolved at runtime.

This unfortunately is no longer true for the two other functions, the most interesting, which are called by the `ap_hook_handler` hook and by `ap_register_input_filter`.

In the first case we are dealing with a clearly more complex function that makes extensive use of encrypted strings *(img 6-7)*.

Image 6

Function performed when the module is invoked

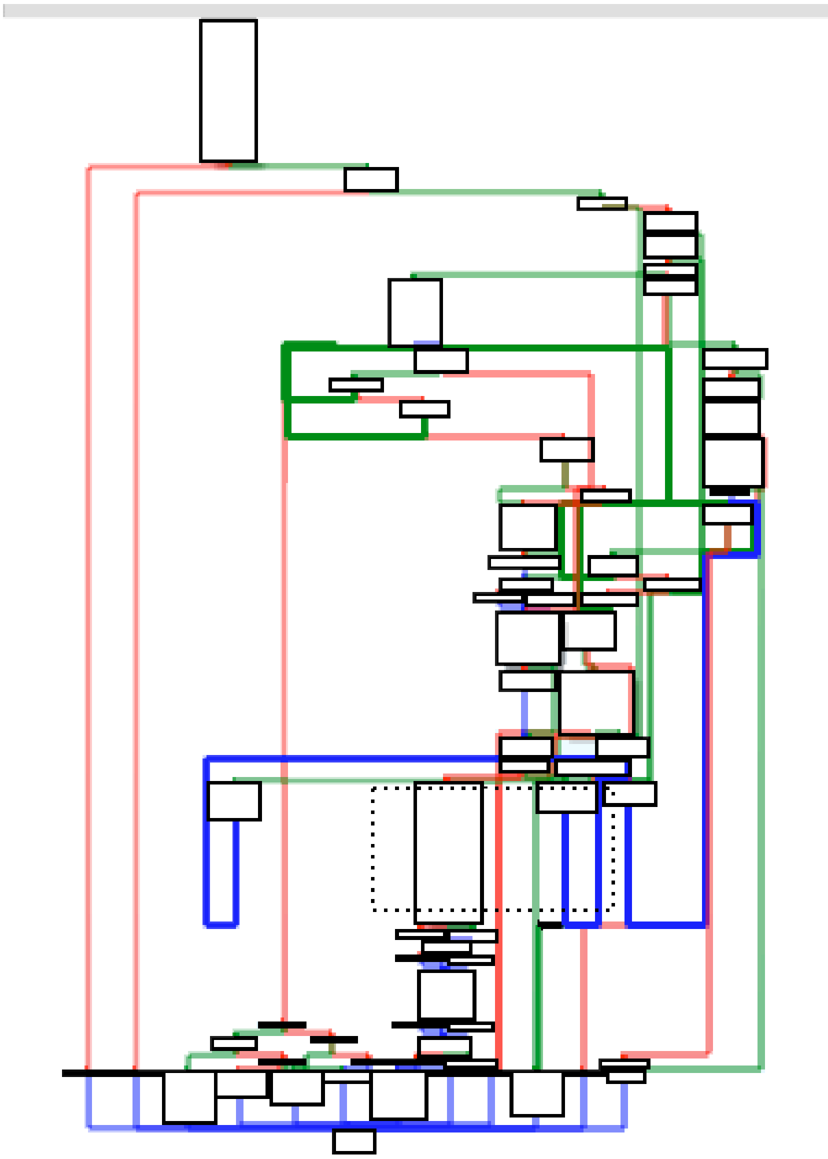


Image 7

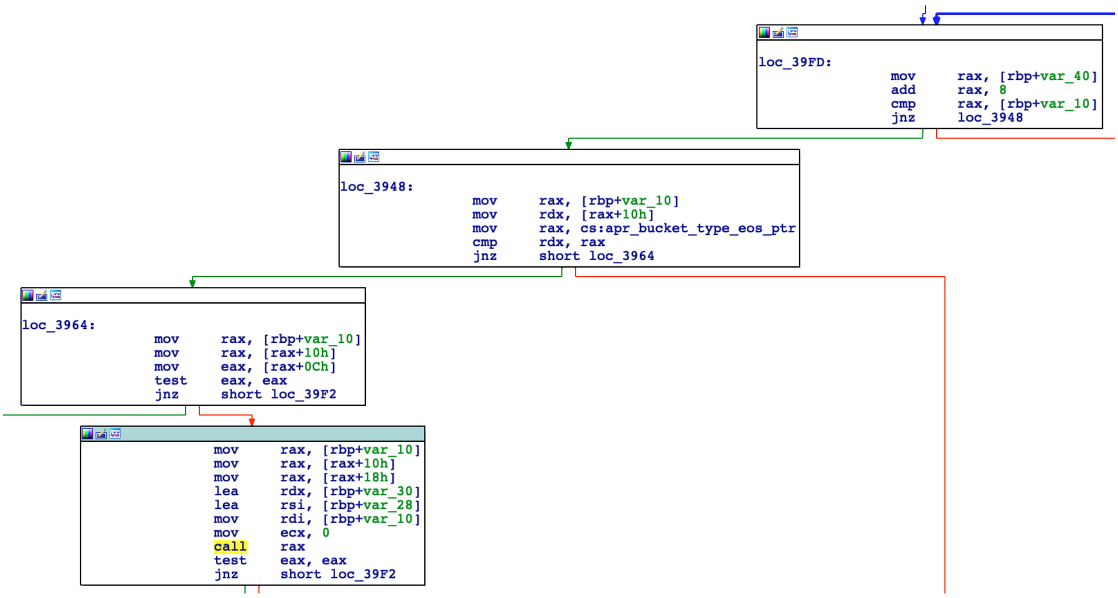
Encrypted strings

Address	Length	Type	String
.data:000000000020845C	00000005	C	cOW\x1Bk
.data:000000000020846A	00000007	C	IDl;tWz
.data:000000000020849A	00000006	C	]?#G:A
.data:00000000002084BA	00000005	C	mktd\\
.data:00000000002084D1	00000005	C	R\v{e_
.data:00000000002084F2	00000006	C	`7*1HM
.data:00000000002085FA	00000006	C	'BWbYo
.data:0000000000208774	00000005	C	<&Pd3
.data:0000000000208853	00000005	C	|l~hY
.data:00000000002088F0	00000005	C	BkirE
.data:0000000000208910	00000005	C	0Un\n(
.data:0000000000208938	00000005	C	]\"]AB
.data:0000000000208975	00000006	C	(\n5\rTj
.data:00000000002089BF	00000005	C	ojE$M
.data:0000000000208AB3	00000005	C	Wu?^n
.data:0000000000208AFF	00000007	C	ui<a}sN
.data:0000000000208B35	00000005	C	Eis'o
.data:0000000000208BF4	00000007	C	}Ou!H4*
.data:0000000000208C6E	00000005	C	A/!>p
.data:0000000000208D0B	00000008	C	>\b.:aFbn
.data:0000000000208DC2	00000005	C	bblYL
.data:0000000000208FB5	00000006	C	-i;RZb
.data:0000000000208FC6	00000006	C	T@},u\x1B
.data:0000000000208FF5	00000005	C	PlOBo
.data:0000000000209063	00000005	C	+?*i
.data:00000000002091C0	00000005	C	b^[QK
.data:00000000002092F2	00000005	C	S%ZA[
.data:0000000000209319	00000005	C	$cq0Z
.data:0000000000209440	00000008	C	Dz27Dz27
.data:00000000002094A8	00000005	C	22PA

In the second case we find calls to functions dynamically resolved at runtime like this one highlighted below: the CALL instruction is resolved at runtime by calling a memory address placed in the RAX register *(img 8)*.

Image 8

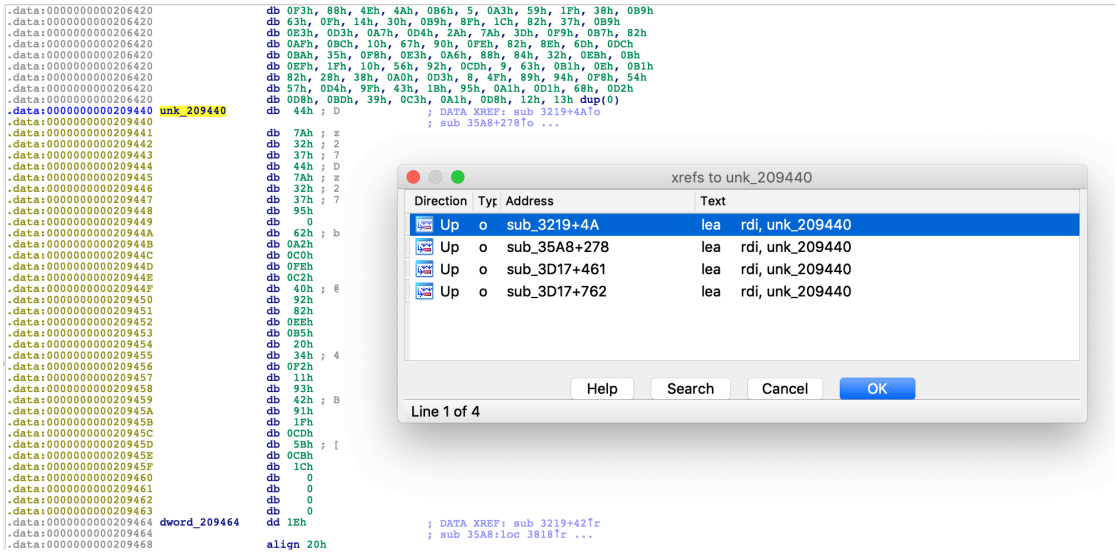
Runtime resolution of the CALL instruction



Before tackling the reversing of these two functions we decide to change approach. Analyzing the encrypted strings we find one that we highlighted earlier, **Dz27Dz27**, which has an important feature *(img 9)*.

Image 9

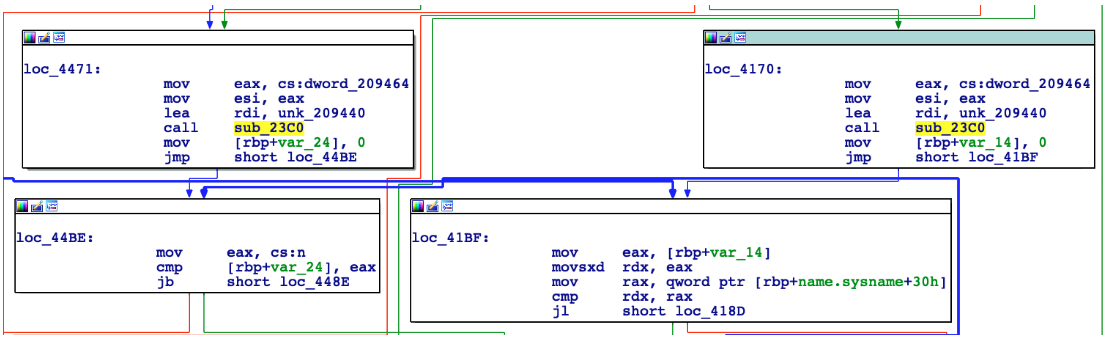
Encrypted string



It is located at a very precise address within the binary and is called several times within the previous two functions that we have decided not to analyze. For example, we find two calls in the function **sub_3D17** *(img 10)*.

Image 10

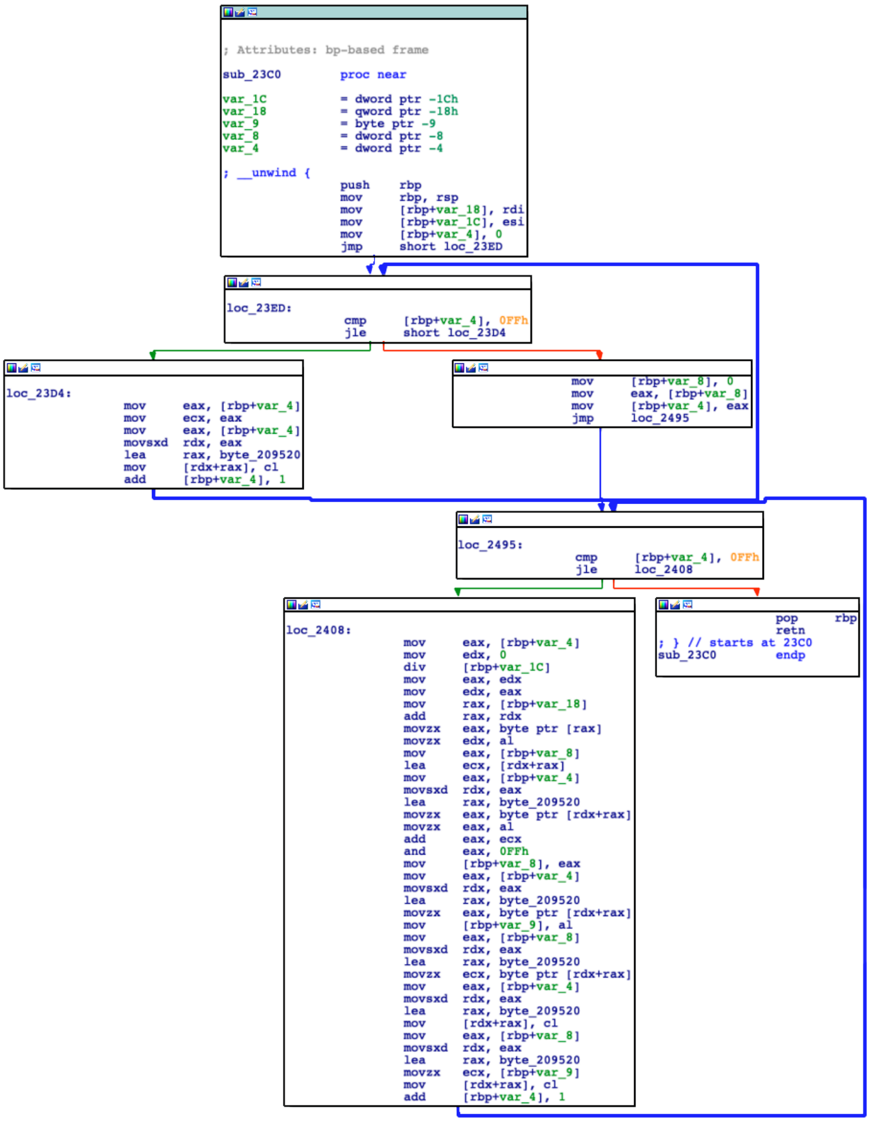
Functions that call the string



They are two locations, **loc_4471** and **loc_4170, which use exactly the same data**. Very interesting is also the fact that the **dword_209464** is positioned exactly after the string highlighted before, almost as if they were the declarations of two variables closely related in some way. Curiosity obviously pushes us to check out what this function does *(img 11)*.

Image 11

The encryption function



In this shape it does not tell us much. However, if we see the decompiled code *(img 12)*.

Image 12

Decompiled function code

```
_BYTE *__fastcall sub_23C0(__int64 a1, unsigned int a2)
{
  _BYTE *result; // rax
  char v3; // ST17_1
  int v4; // [rsp+14h] [rbp-8h]
  signed int i; // [rsp+18h] [rbp-4h]
  signed int j; // [rsp+18h] [rbp-4h]

  for ( i = 0; i <= 255; ++i )
    byte_209520[i] = i;
  LOBYTE(v4) = 0;
  result = 0LL;
  for ( j = 0; j <= 255; ++j )
  {
    v4 = (unsigned __int8)(*(_BYTE *)(j % a2 + a1) + v4 + byte_209520[j]);
    v3 = byte_209520[j];
    byte_209520[j] = byte_209520[v4];
    result = byte_209520;
    byte_209520[v4] = v3;
  }
  return result;
}
```

Those who deal with malware reverse engineering and encryption algorithms will have understood that in cases like this - nine times out of ten - this is the RC4 encryption algorithm. This is in fact the part of the algorithm known as **KSA (Key Scheduling Algorithm)** *(img 13)*.

Image 13
KSA (Key Scheduling Algorithm)
Source here

Key-scheduling algorithm (KSA) [edit]

The key-scheduling algorithm is used to initialize the permutation in the array "S". "keylength" is defined as the number of bytes in the key and can be in the range 1 ≤ keylength ≤ 256, typically between 5 and 16, corresponding to a key length of 40 – 128 bits. First, the array "S" is initialized to the identity permutation. S is then processed for 256 iterations in a similar way to the main PRGA, but also mixes in bytes of the key at the same time.

```
for i from 0 to 255
    S[i] := i
endfor
j := 0
for i from 0 to 255
    j := (j + S[i] + key[i mod keylength]) mod 256
    swap values of S[i] and S[j]
endfor
```

The **key** and its **length** are identified as arguments of the function. Once this new information is obtained, we are therefore able to trace where this algorithm is used to encrypt and decrypt data saved statically in the binary.
We find the first reference in this interesting function in which we were also able to identify the second part of the RC4 algorithm, the one known as **Pseudo-random generation algorithm (PRGA)** *(img 14)*.

Image 14
Custom implementation of RC4 encryption
Source here

```
; Attributes: bp-based frame

sub_3219        proc near

var_40          = qword ptr -40h
var_38          = qword ptr -38h
var_28          = qword ptr -28h
var_20          = qword ptr -20h
counter         = dword ptr -14h

; __unwind {
                push    rbp
                mov     rbp, rsp
                push    r12
                push    rbx
                sub     rsp, 30h
                mov     [rbp+var_38], rdi
                mov     [rbp+var_40], rsi
                mov     [rbp+var_20], 0
                mov     rax, [rbp+var_38]
                mov     esi, 170h
                mov     rdi, rax
                call    _apr_palloc
                mov     edx, 170h ; n
                mov     esi, 0 ; c
                mov     rdi, rax ; s
                call    _memset
                mov     [rbp+var_28], rax
                mov     eax, cs:RC4_KeyLength
                mov     esi, eax
                lea     rdi, RC4_Key
                call    KSA_Function
                mov     [rbp+counter], 0
                jmp     short loc_32A5
```

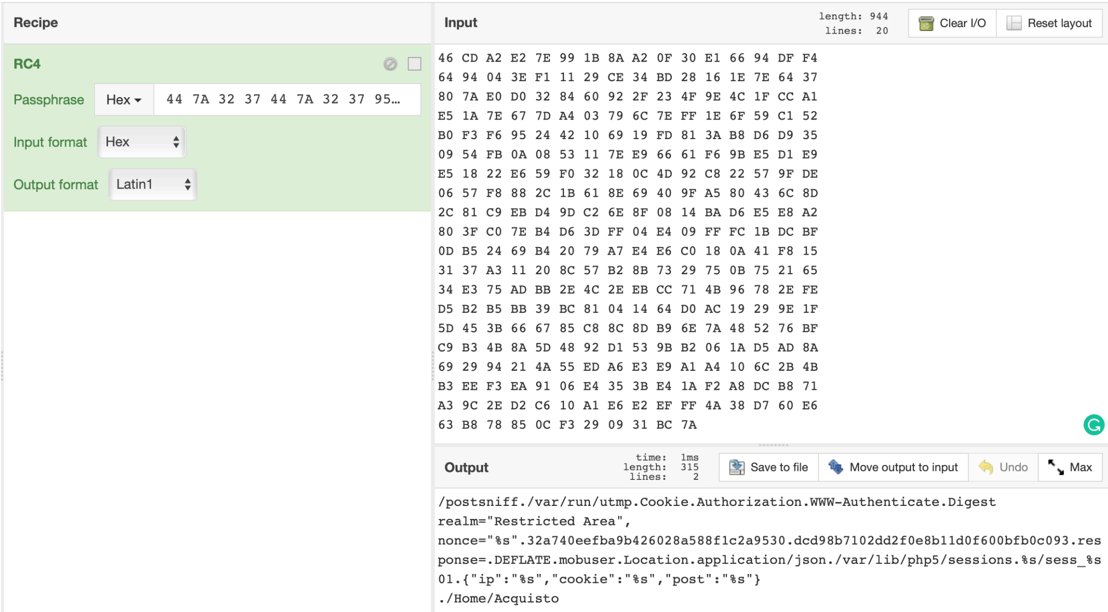
```
loc_32A5:
                mov     eax, cs:EncryptedBufferLength
                cmp     [rbp+counter], eax
                jb      short loc_3278
```

```
loc_3278:
                mov     edx, [rbp+counter]
                mov     rax, [rbp+var_28]
                lea     rbx, [rdx+rax]
                mov     edx, [rbp+counter]
                lea     rax, EncryptedBuffer
                movzx   r12d, byte ptr [rdx+rax]
                mov     eax, [rbp+counter]
                mov     edi, eax
                call    PRGA_Function
                xor     eax, r12d
                mov     [rbx], al
                add     [rbp+counter], 1
```

```
                mov     eax, 0
                call    sub_2558
                mov     rax, [rbp+var_28]
                mov     [rbp+var_20], rax
                mov     rax, [rbp+var_20]
                add     rsp, 30h
                pop     rbx
                pop     r12
                pop     rbp
                retn
; } // starts at 3219
sub_3219        endp
```

Having therefore the RC4 key and the length of the encrypted buffer we can proceed to decrypt the buffer. To do this we will use **CyberChef** *(img 15)*.

Image 15
CyberChef
Source here



These evidences have been **disruptive** for the outcome of the incident response and resolution of the case. Analysing these strings we can surely proceed forward in the forensic activity of the machine and therefore obtain Indicators Of Compromise (IoC).

Another very important indication of **how specifically the module was built for the Customer's environment is the presence of the string /Home/Acquisto**. This information fits exactly with what we have understood from the static analysis: the regular expression created during the initialization of the module would seem to look for the match with this string which, incidentally, is exactly the URL that deals with finalizing the monetary transaction where the user enters the data for payment.

The RC4 algorithm is used in many other functions within the module: each data is never in clear text but always encrypted. However, having the encryption key we can proceed a little faster. In particular, a full-bodied buffer inside the track aroused the curiosity of the reverse engineering team *(fig.16)*.

Image 16

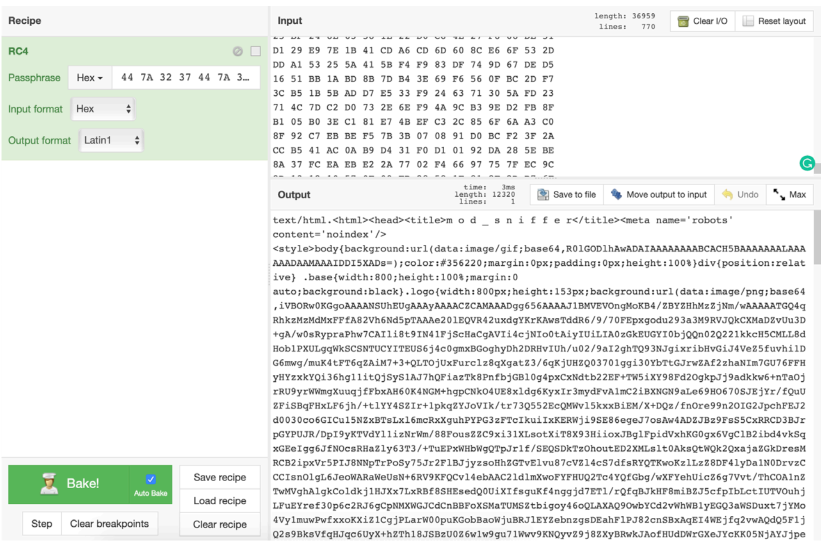
Encrypted buffer

```
EncryptedPostsniffPageLength dd 300Dh    ; DATA XREF: sub 3D17+567↑r
                                         ; sub 3D17+56F↑r ...
                align 20h
; _BYTE EncryptedPostsniffPage[12320]
EncryptedPostsniffPage db 1Dh, 0D8h, 0B5h, 0E5h, 25h, 82h, 1, 8Eh, 0A8h, 69h
                                         ; DATA XREF: sub 3D17+788↑o
                db 0Ch, 0A6h, 64h, 98h, 0C1h, 0E5h, 2Ah, 89h, 0Fh, 70h
                db 0E0h, 5Bh, 78h, 0CAh, 5Dh, 8Ah, 2Bh, 1Ch, 4Bh, 7Ah
                db 21h, 58h, 0E1h, 6Bh, 0B4h, 0E7h, 7Dh, 85h, 29h, 86h
                db 6Eh, 3Eh, 6, 97h, 2, 79h, 0BBh, 93h, 92h, 45h, 3, 3Dh
                db 7Dh, 0A5h, 12h, 7Bh, 7Dh, 29h, 0A0h, 12h, 7Eh, 48h
                db 0A0h, 36h, 0B7h, 0F5h, 0FEh, 83h, 6Dh, 45h, 10h, 63h
                db 1Ah, 0FEh, 98h, 74h, 0BDh, 0A4h, 0DFh, 29h, 13h, 52h
                db 0F7h, 7, 8, 0Bh, 52h, 30h, 0C7h, 7Dh, 6Ah, 0F3h, 0DCh
                db 0B1h, 0D6h, 0A8h, 0B4h, 4Ah, 32h, 0F7h, 1Dh, 0BEh, 72h
                db 55h, 4Ch, 22h, 0C5h, 83h, 38h, 2, 0CAh, 8Dh, 8, 55h
                db 0ECh, 85h, 38h, 4Ch, 67h, 80h, 2Eh, 4, 0C3h, 0BFh, 0DCh
                db 43h, 2Dh, 0D4h, 2Eh, 8Eh, 95h, 0E9h, 81h, 99h, 0D4h
                db 3Ch, 0D5h, 5Eh, 2Fh, 0BCh, 0D4h, 0F2h, 0B4h, 0ACh, 0D6h
                db 24h, 0A3h, 7Eh, 0EAh, 0F5h, 16h, 89h, 0Eh, 0BCh, 2Dh
                db 0B0h, 0DFh, 6Eh, 0ACh, 92h, 7Ch, 92h, 53h, 18h, 0C5h
                db 3, 5Eh, 84h, 96h, 0C6h, 0B1h, 62h, 71h, 74h, 0C8h, 31h
                db 3, 6, 8Dh, 3Eh, 12h, 0A8h, 26h, 0F3h, 8Eh, 77h, 2Eh
                db 78h, 0Bh, 65h, 25h, 24h, 14h, 0CDh, 56h, 99h, 81h, 0Fh
                db 7Ah, 67h, 92h, 0FBh, 53h, 6Eh, 91h, 2Ch, 68h, 0ABh
                db 0B6h, 0BCh, 0A9h, 0A4h, 27h, 0EFh, 0C1h, 56h, 55h, 3Bh
                db 8Dh, 0F0h, 6, 78h, 80h, 11h, 41h, 22h, 7Dh, 7Eh, 3Ch
                db 0C7h, 97h, 98h, 0DFh, 0ABh, 20h, 6Eh, 44h, 4Bh, 2Dh
                db 0F7h, 80h, 0E6h, 48h, 81h, 0Fh, 4Fh, 99h, 0CBh, 34h
                db 0D6h, 0B5h, 13h, 58h, 80h, 0EEh, 0DCh, 4Bh, 68h, 8Eh
                db 67h, 0, 25h, 0F9h, 0F7h, 0E3h, 0EDh, 0EAh, 0F1h, 5Ch
                db 73h, 2Ah, 0Ch, 0F3h, 0ADh, 0E4h, 0ECh, 88h, 8, 0F0h
                db 70h, 37h, 0BCh, 59h, 0A4h, 0BEh, 85h, 0E3h, 3Ah, 0B7h
                db 87h, 35h, 9Ch, 0DCh, 1Ah, 0B3h, 0F8h, 0F9h, 0A8h, 0EBh
                db 27h, 50h, 8Ch, 12h, 0B8h, 3Eh, 0EDh, 72h, 0FFh, 2, 0E3h
                db 2Eh, 7, 2Bh, 0A6h, 2Fh, 9, 78h, 0C3h, 2, 0FCh, 0C7h
                db 59h, 0E4h, 0BEh, 14h, 0A0h, 0A7h, 15h, 35h, 86h, 0ADh
                db 31h, 7Ah, 7Fh, 65h, 0BAh, 0D2h, 66h, 0D7h, 7, 0C3h
                db 82h, 0B2h, 2 dup(67h), 0B4h, 59h, 9Ch, 76h, 5Dh, 3Eh
                db 7Fh, 0Ah, 4Fh, 48h, 34h, 0CDh, 0F3h, 99h, 0ACh, 71h
                db 0D5h, 19h, 65h, 2, 49h, 0E2h, 0E8h, 0E1h, 14h, 0FBh
                db 0C7h, 77h, 0EDh, 91h, 0ABh, 91h, 0, 4, 3, 9, 0A3h, 0FAh
                db 0EAh, 2Bh, 0F8h, 76h, 61h, 15h, 59h, 34h, 0A0h, 0D7h
                db 1Fh, 37h, 50h, 98h, 8Bh, 3Fh, 4Fh, 90h, 18h, 4Ah, 32h
                db 36h, 6Eh, 6Dh, 0B5h, 2 dup(40h), 7Dh, 27h, 0B6h, 7
                db 11h, 30h, 0FCh, 0D0h, 4Eh, 18h, 67h, 7Bh, 18h, 79h
                db 9Dh, 18h, 36h, 0B9h, 96h, 5, 0C7h, 5Eh, 4Dh, 96h, 0A3h
                db 0C9h, 14h, 0E3h, 88h, 0F7h, 0B4h, 5, 0A2h, 49h, 0DFh
                db 46h, 38h, 37h, 36h, 0FFh, 71h, 0D7h, 6Eh, 1Fh, 9Eh
                db 0A0h, 8Fh, 0B9h, 0A2h, 0D2h, 0D9h, 0AAh, 4Fh, 0E8h
                db 14h, 8Ah, 0FFh, 0E0h, 93h, 4, 9Bh, 98h, 25h, 7Bh, 0EBh
                db 0BEh, 0EAh, 0D0h, 7Eh, 34h, 7Fh, 0FAh, 2Ah, 83h, 7Ah
                db 63h, 5Ch, 4Dh, 80h, 8Fh, 0C5h, 53h, 59h, 72h, 6Eh, 0F4h
                db 50h, 34h, 0Bh, 4Bh, 64h, 0Fh, 3Dh, 0B1h, 0E5h, 22h
                db 0, 0B2h, 11h, 1Eh, 0DAh, 4Eh, 4Ch, 87h, 1Ch, 15h, 0F5h
                db 92h, 0D9h, 0CDh, 18h, 76h, 0E8h, 11h, 4Bh, 0C1h, 92h
                db 4, 9Ah, 26h, 0B1h, 0B4h, 80h, 9Fh, 1, 4, 47h, 7Bh, 9Ah
                db 96h, 3Ah, 5, 0A7h, 28h, 0D5h, 94h, 4Bh, 0E7h, 25h, 90h
                db 68h, 38h, 13h, 22h, 85h, 9Fh, 6Fh, 19h, 8Ch, 6, 0EDh
                db 0CFh, 0F0h, 36h, 9Fh, 0F3h, 4Ah, 0Fh, 0FDh, 27h, 0BDh
                db 0CAh, 7Eh, 8Bh, 0F8h, 62h, 65h, 13h, 0DAh, 0F4h, 0FAh
                db 0B0h, 31h, 2Dh, 8Bh, 0AAh, 2 dup(82h), 0C6h, 31h, 19h
```

A buffer of around 12KB that we can decipher with the same methodology as before *(img 17)*.

Image 17

Decrypted buffer



An html page, saved inside the module, showing a title **mod_sniffer** and an image called **modIframe** *(img 18)*.

Image 18
modIframe

Apart from the nice subtitle the page shows some interesting information: there are in fact variables that are obviously resolved at runtime like the kernel version or uptime and others of which, at the moment, we do not know the meaning.

The module is still in the analysis phase.

We share the hashes that identify it:

MD5
1720aca23d81e0aa6fa28096781294c3

SHA-1
df454026aac01ad7e394c9f5c2bfdb12fea9a0e0

SHA-256
1c55ffee91e8d8d7a1b4a1290d92a58c4da0c509d5d8d2741cec7f4cf6a099bd

Author

Mario Ciccarelli

Head of IR Team Yarix - Digital Security Division Var Group

@Kartone

Special thanks to

Alessandro Amadori

Solution Specialist - Digital Security Division Var Group

YARIX
a vargroup company

YARIX
a vargroup company