The Kutaki Malware Bypasses Gateways to Steal Users' Credentials

cofense.com/kutaki-malware-bypasses-gateways-steal-users-credentials/

Cofense January 21, 2019

CISO Summary

It's a case of hiding in plain sight. Cofense™ recently found a phishing campaign that hides the Kutaki malware in a legitimate application to bypass email gateways and harvest users' credentials.

A data stealer, Kutaki uses old-school techniques to detect sandboxes and debugging, but don't underestimate it—Kutaki works quite well against unhardened virtual machines and other analysis devices. By backdooring a legitimate application, it can fool unsophisticated detection methodologies.

Learn how Cofense Intelligence™ keeps IT teams ahead of the latest phishing and malware threats like this new campaign.

Full Details

Cofense Intelligence recently uncovered a small-scale phishing campaign delivering a sample of the Kutaki information stealer and keylogger that was hidden inside a legitimate Visual Basic application and delivered as an OLE package within a weaponized Office document.

Kutaki uses a series of anti-virtualization and anti-analysis techniques that were ostensibly copied verbatim from a series of blogs dating back to 2010-2011. Kutaki – a data stealer – is capable of harvesting Input data directly from keyboards, mice, clipboards, microphones, and screens (in the form of screenshots). The campaign observed by Cofense Intelligence also saw Kutaki retrieve a copy of SecurityXploded's BrowserPasswordDump utility by dropping and executing a copy of cURL for Windows.

Despite the evasion techniques being antiquated, they are somewhat successful against causal observation and analysis.

Hiding in Plain Sight: Obfuscation

This variant of Kutaki uses the source code as a Visual Basic training app to hide its malicious content. By backdooring an ostensibly simple training app, it attempts to exploit any potential whitelisting or simply bypass static signatures.

Figure 1 shows the backdoored application as a project breakdown. Figure 2 is a closer view of the procedures.

Project

```
Type=Exe
Reference=*\G{00020430-0000-0000-C000-000000000046}#2.0#0#..\..\..\..\WINDOWS\SYSTEM32\STDOLE2.TLB#OLE Automation
Object={00000000-0000-0000-0000-000000000000}#0; C:\Windows\SysWow64\MSDBRPTR.DLL
Object={00000000-0000-0000-0000-000000000000}#0; C:\Program Files (x86)\Common Files\designer\MSDERUN.DLL
Object={00000000-0000-0000-0000-000000000000}#0; MSHFLXGD.OCX
Object={EAB22AC0-30C1-11CF-A7EB-0000C05BAE0B}#1.1#0; ieframe.dll
Form=frmcustomer.frm
Form=frmtabledet.frm
Form=frmwaiterdet.frm
Form=frmassets.frm
Form=frmbilling.frm
Form=ff.frm
Form=frmLogin.frm
Module=chee; chee.bas
Module=saamneao; saamneao.bas
Module=dewani; dewani.bas
Module=ende; ende.bas
Module=JSON; JSON.bas
Module=modPlaySound; modPlaySound.bas
Module=Module1dd; Module1dd.bas
Module=Module4; Module4.bas
Module=stringbroda; stringbroda.bas
Module=tero; tero.bas
Class=cJSONScript; cJSONScript.cls
Class=cStringBuilder; cStringBuilder.cls
Module=khi; khi.bas
Form=frmdishdet.frm
Form=frmdailyexpense.frm
Form=frmswaiter.frm
Form=frmsdish.frm
Form=frmscustomer.frm
Form=Mdimain.frm
Form=frmabout.frm
Designer=datrepdish.dsr
Form=frmdelwaiter.frm
Form=frmdelcust.frm
Form=frmdeldish.frm
Form=frmdeltable.frm
Form=frmdelassets.frm
Module=Module1; Module1.bas
Form=frmcashier.frm
Designer=datrepbill.dsr
Designer=datrepcustomer.dsr
Designer=datrepwaitersal.dsr
Designer=datrepassets.dsr
Designer=datenvrestaurant.dsr
Form=frmSplash.frm
Designer=datrepdailyexp.dsr
Form=terabap.frm
```

Figure 1: the project breakdown

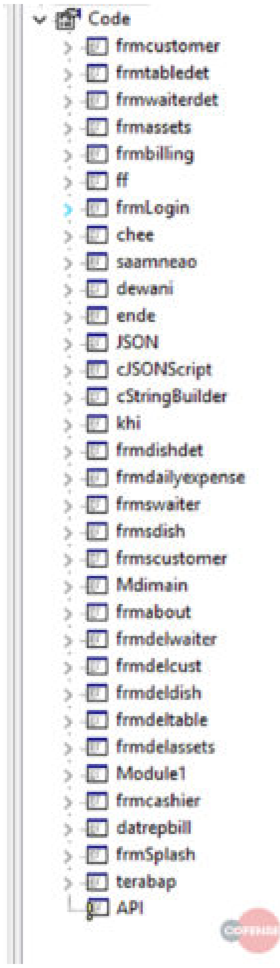


Figure 2: The code sections (procedures) present in the project.

Even for non-programmers, there are certain procedure names that seem to be wildly misplaced. Indeed, we can see a close (but not quite complete) correlation between the Forms – which are GUI elements – and the procedures that power them. Figure 3 demonstrates this mapping.

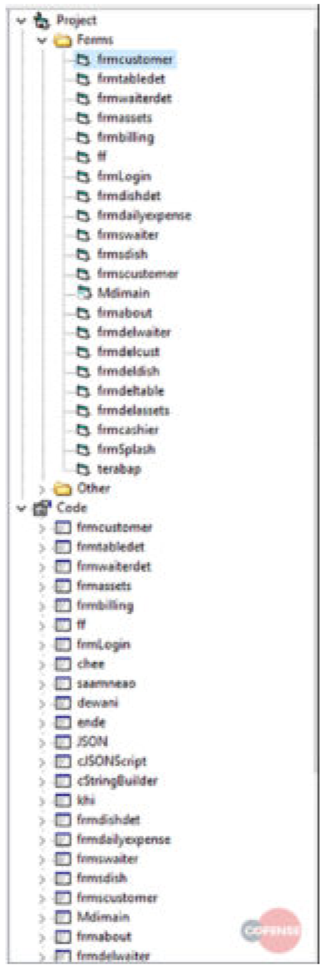


Figure 3: Form elements relative to their procedure (code) counterparts.

Final proof that this application has been backdoored can be found by inspecting the procedures. Figure 4 shows legitimate procedures compared with those that have been injected.

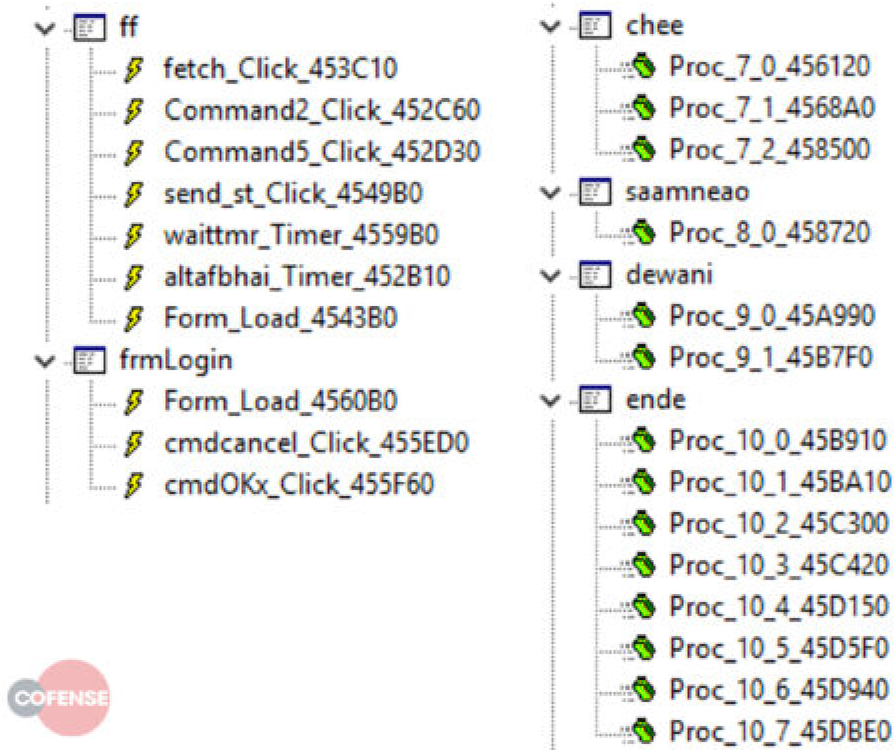


Figure 4: “ff” and “frmLogin” are original procedures. “chee”, “saamneao”, “dewani” and “ende” are injected.

Not only do we see a discrepancy between the naming conventions – most legitimate procedures here begin with ‘frm’ – but we can also intuit the random names assigned to the injected procedures. Further, the functions – those found within the injected procedures – have unresolvable names, so they’re simply assigned one by the decompiler.

Diving into some of this injected code yields even more obfuscation. Strings within the binary are reversed and decoded using the rtcStrReverse function. Figure 5 shows an example of such obfuscation.

```
push    ebp
mov     ebp, esp
sub     esp, 8
push    offset __vbaExceptHandler
mov     eax, large fs:0
push    eax
mov     large fs:0, esp
sub     esp, 64h
push    ebx
push    esi
push    edi
mov     [ebp+var_8], esp
mov     [ebp+var_4], offset dword_402778
mov     ebx, ds:rtcStrReverse
xor     edi, edi
push    offset off_41A57C
mov     [ebp+var_14], edi
mov     [ebp+var_18], edi
mov     [ebp+var_1C], edi
mov     [ebp+var_20], edi
mov     [ebp+var_24], edi
mov     [ebp+var_28], edi
mov     [ebp+var_2C], edi
mov     [ebp+var_30], edi
mov     [ebp+var_34], edi
mov     [ebp+var_44], edi
mov     [ebp+var_54], edi
mov     [ebp+var_5C], edi
call    ebx ; rtcStrReverse
mov     esi, ds:__vbaStrMove
mov     edx, eax
lea     ecx, [ebp+var_14]
call    esi ; __vbaStrMove
push    offset aMuneKsidSecivr ; "munE\\ksiD\\secivreS\\100teSlortnoC\\ME"...
call    ebx ; rtcStrReverse
```

Figure 5: 3 instances of rtcStrReverse being used to deobfuscate stored strings.

Similar string obfuscation techniques can be found masking suspicious API calls. Figure 6 shows the obfuscation of Sleep and ShellExecuteA strings.

Figure 6: Sleep and ShellExecuteA strings.

These strings are part of a small struct used by DllFunctionCall – a method by which Visual Basic applications can retrieve the addresses of functions from specific DLLs. The struct looks something like this:

typedef struct _DllFunctionCallDataStruct {

void * lpLibName;

void * lpExportName;

} DllFunctionCallDataStruct;

We can see how this structure maps to what we see in the disassembly in figure 6. All calls to DLLFunctionCall are wrapped in identical snippets, as demonstrated in Figure 7.

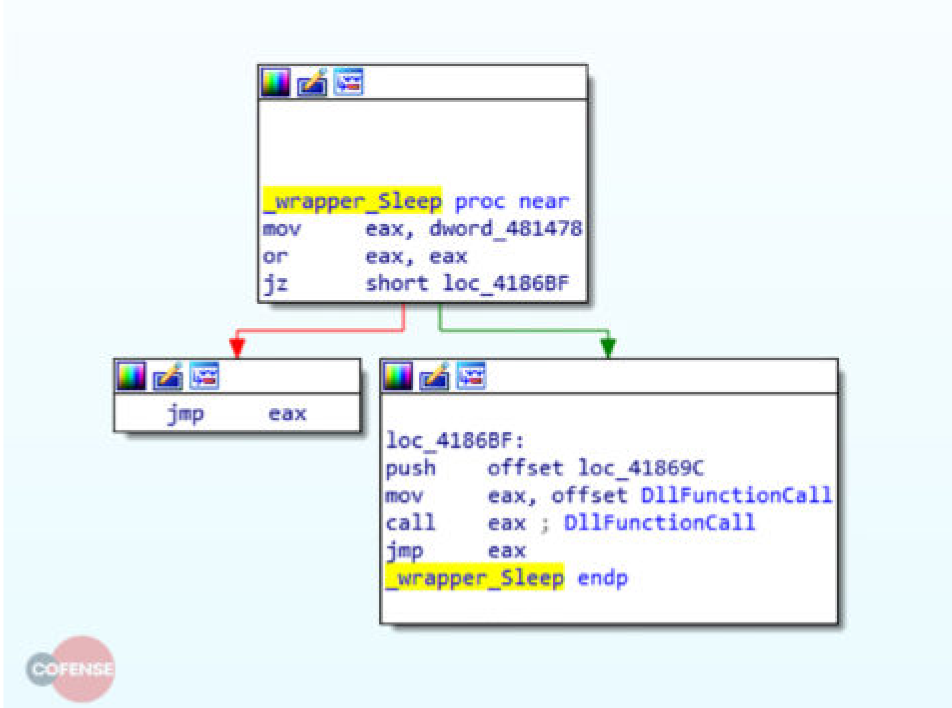


Figure 7: a typical wrapper for calls to DllFunctionCall.

After careful analysis, we find that 18 high-value API calls are obfuscated in this manner. Figure 8 details these.

Sleep	.text
ShellExecuteA	.text
GetWindowsDirectoryA	.text
GetComputerNameA	.text
GetKeyState	.text
GetForegroundWindow	.text
GetWindowTextA	.text
GetWindowTextLengthA	.text
GetAsyncKeyState	.text
CreateToolhelp32Snapshot	.text
Module32First	.text
Module32Next	.text
GetWindowsDirectoryA_0	.text
GetCurrentProcessId	.text
RegOpenKeyExA	.text
RegQueryValueExA	.text
RegCloseKey	.text
rtlMoveMemory	.text

Figure 8: De-obfuscated API calls used by Kutaki to perform some of its malicious activity.

Anti-Virtualization

Kutaki employs some basic checks and comparisons to identify whether it is executing within a virtualized environment. The first of these involves reading the HKLM\System\CurrentControlSet\Services\Disk\Enum registry key and comparing the returned string against a list of “undesirable” strings. Figure 9 details the read of this key.

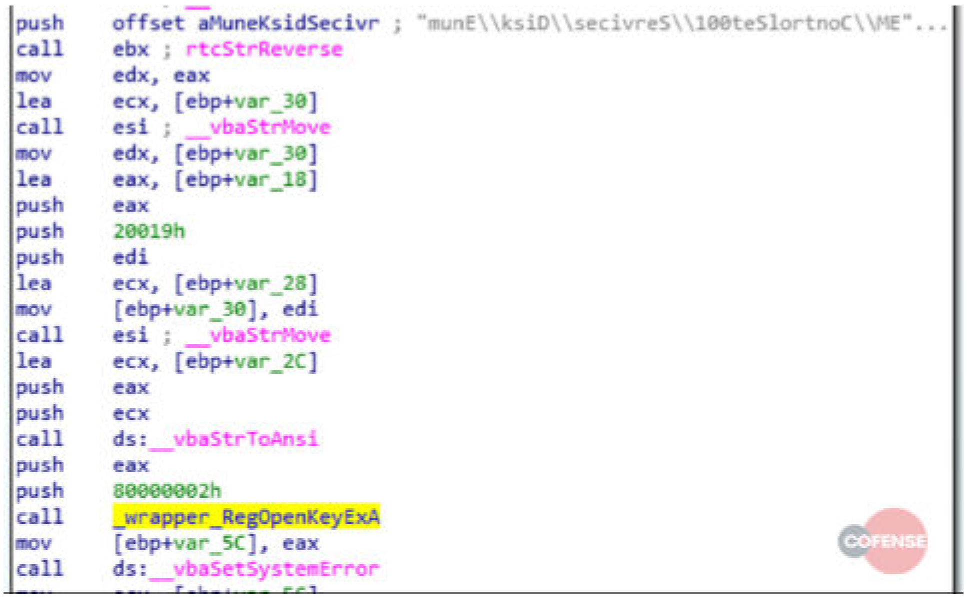


Figure 9: Kutaki reads disk metadata from the registry.

This registry key contains information about the disks present on the machine. The first disk is stored in a value named “0”, the second in a value named “1” and so on. In the instance of this analysis VM, the value 0 contains the data observed in figure 10.

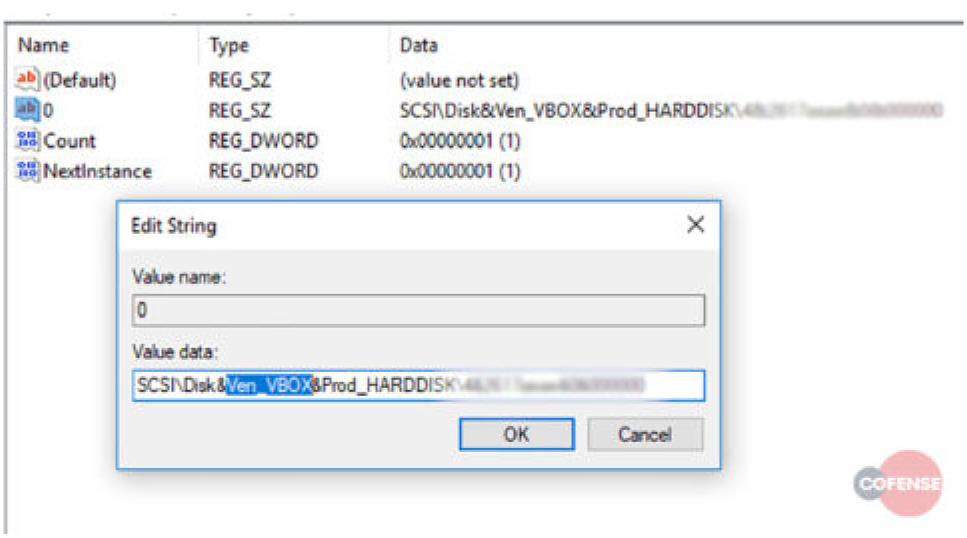


Figure 10: Example data from the Disks\Enum registry key.

The highlighted text shows that the disk belongs to a VirtualBox VM. Figures 11 and 12 show two different string comparisons, attempting to identify different types of virtual machines present. Figure 13 is all the strings Kutaki will compare against.

```
push    eax
push    offset aLautriv ; "*LAUTRIV*"
call    ebx ; rtcStrReverse
mov     edx, eax
lea     ecx, [ebp+var_28]
call    esi ; __vbaStrMove
push    eax
call    ds:__vbaStrLike
xor     ecx, ecx
cmp     ax, 0FFFFh
setz    cl
neg     ecx
mov     [ebp+var_58], ecx
lea     ecx, [ebp+var_28]
call    ds:__vbaFreeStr
cmp     word ptr [ebp+var_58], di
jz      short loc_4674A4
```

Figure 11: Check if the registry value contains “VIRTUAL” anywhere within the string.

```
loc_4674C1:
mov     ecx, [ebp+var_24]
push    ecx
push    offset aXobv    ; "*XOBV*"
call    ebx ; rtcStrReverse
mov     edx, eax
lea     ecx, [ebp+var_28]
call    esi ; __vbaStrMove
push    eax
call    ds:__vbaStrLike
xor     edx, edx
cmp     ax, 0FFFFh
setz    dl
neg     edx
lea     ecx, [ebp+var_28]
mov     esi, edx
call    ds:__vbaFreeStr
cmp     si, di
jz      short loc_4674FC
```

Figure 12: Check if the registry value contains “VBOX” anywhere in the string.

```
.text:0041A584 aWmv:
.text:0041A584                 text "UTF-16LE", 'WMV*',0
.text:0041A58E                 align 10h
.text:0041A590 aN_0:
.text:0041A590                 text "UTF-16LE", 'N',0
.text:0041A594 aMuneKsidSecivr:                        ; DATA XREF: Anti_VM_Enum_Disks+69↓o
.text:0041A594                 text "UTF-16LE", 'munE\ksiD\secivreS\100teSlortnoC\METSYS',0
.text:0041A5E4                 dd 12h
.text:0041A5E8 aLautriv:                               ; DATA XREF: Anti_VM_Enum_Disks+1E9↓o
.text:0041A5E8                 text "UTF-16LE", '*LAUTRIV*',0
.text:0041A5FC                 dd 0Ch
.text:0041A600 aXobv:                                  ; DATA XREF: Anti_VM_Enum_Disks+245↓o
.text:0041A600                 text "UTF-16LE", '*XOBV*',0
.text:0041A60E                 align 10h
.text:0041A610                 dd 16h
.text:0041A614 aLldLldeibs:                            ; DATA XREF: sub_467570+203↓o
.text:0041A614                 text "UTF-16LE", 'lld.lldeibs',0
.text:0041A62C                 dd 16h
.text:0041A630 aLldPlehgbd:                            ; DATA XREF: sub_467570+318↓o
.text:0041A630                 text "UTF-16LE", 'lld.plehgbd',0
.text:0041A648 aR:
.text:0041A648                 text "UTF-16LE", 'R',0
.text:0041A64C aNoisrevtnerruc:                        ; DATA XREF: sub_467570+3F0↓o
.text:0041A64C                 text "UTF-16LE", 'noisreVtnerruC\swodniW\tfosorciM\erawtfoS',0
.text:0041A6A0                 text "UTF-16LE", '.',0
.text:0041A6A4 a41622559924873:                        ; DATA XREF: sub_467570+570↓o
.text:0041A6A4                 text "UTF-16LE", '41622-5599248-733-78467',0
.text:0041A6D4                 text "UTF-16LE", '.',0
.text:0041A6D8 a01532730771344:                        ; DATA XREF: sub_467570+5C5↓o
.text:0041A6D8                 text "UTF-16LE", '01532-7307713-446-78467',0
.text:0041A708                 text "UTF-16LE", '.',0
.text:0041A70C a05932460376204:                        ; DATA XREF: sub_467570+617↓o
.text:0041A70C                 text "UTF-16LE", '05932-4603762-046-47255',0
.text:0041A73C                 text "UTF-16LE", 8,0
.text:0041A740 aPtth:                                  ; DATA XREF: sub_468010+C3↓o
.text:0041A740                 text "UTF-16LE", 'ptth',0
```

Figure 13: Anti-virtualization strings.

The string comparison seen in figure 12 would match the data found in the registry value displayed in Figure 10. Despite the match, Kutaki doesn’t immediately exit, rather it continues with other virtualization checks. Only after all checks are completed will it determine whether execution should continue. Figure 14 shows the execution flow of this concept. The specifics of the results checker are detailed later.

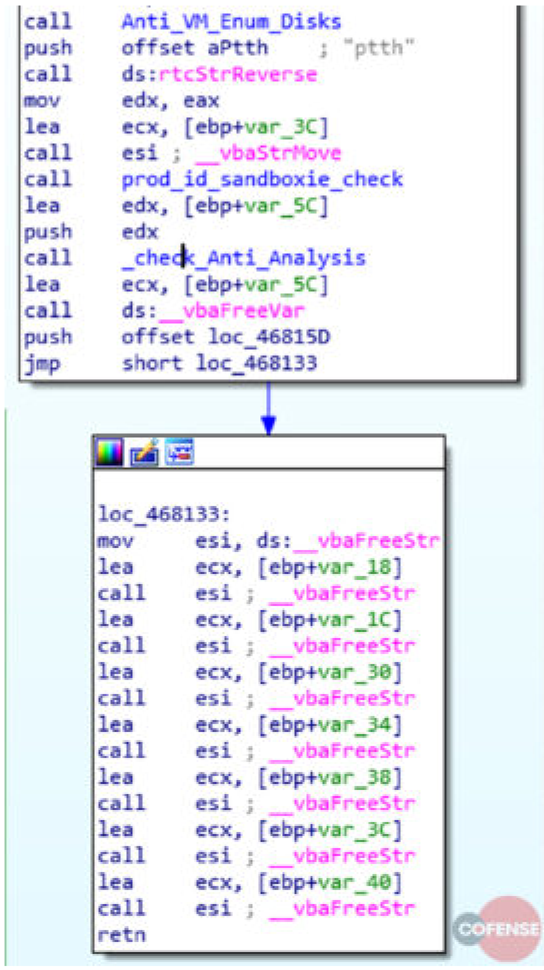


Figure 14: Anti-analysis/virtualization chain.

To supplement the disk checks, Kutaki attempts to determine whether specific modules, which belong to sandboxes and debugging utilities, have been injected into its address space. It achieves this by using a combination of CreateToolhelp32Snapshot, Module32First and Module32Next. These APIs take a snapshot of the running process (including heap, modules, etc.), find the first module, and iterate over subsequent modules mapped to the process, respectively. Figure 15 shows Kutaki setting up the snapshots and retrieving a pointer to the first module.

```
mov     [ebp+var_18], ebx
mov     [ebp+var_A54], ebx
mov     [ebp+var_A5C], ebx
mov     [ebp+var_A60], ebx
mov     [ebp+var_A64], ebx
mov     [ebp+var_A68], ebx
mov     [ebp+var_A6C], ebx
mov     [ebp+var_A7C], ebx
mov     [ebp+var_A8C], ebx
mov     [ebp+var_A9C], ebx
mov     [ebp+var_AAC], ebx
mov     [ebp+var_ABC], ebx
mov     [ebp+var_ACC], ebx
mov     [ebp+th32ProcessID], ebx
mov     [ebp+var_AD4], ebx
call    ds:__vbaFixstrConstruct
call    _warpper_GetCurrentProcessID
mov     esi, ds:__vbaSetSystemError
mov     [ebp+th32ProcessID], eax
call    esi ; __vbaSetSystemError
mov     ecx, [ebp+th32ProcessID]
push    ecx             ; th32ProcessID
push    8               ; dwFlags
call    _warpper_CreateToolhelp32Snapshot
mov     [ebp+var_AD4], eax
call    esi ; __vbaSetSystemError
mov     edi, [ebp+var_AD4]
lea     edx, [ebp+uSize]
lea     eax, [ebp+var_FFC]
push    edx
push    eax
push    offset asc_41957C ; " "
mov     [ebp+lpBuffer], edi
mov     [ebp+uSize], 51Dh
call    ds:__vbaRecUniToAnsi
push    eax
push    edi
call    _wrapper_Module32First
call    esi ; __vbaSetSystemError
lea     ecx, [ebp+var_FFC]
lea     edx, [ebp+uSize]
push    ecx
push    edx
push    offset asc_41957C ; " "
call    ds:__vbaRecAnsiToUni
mov     esi, ds:__vbaStrMove
mov     edi, ds:rtcStrReverse
```

Figure 15: Kutaki setting up a module-identification loop.

Kutaki checks for the existence of sbiedll.dll and dbghelp.dll. These modules belong to Sandboxie and Microsoft, respectively. Figure 16 shows the de-obfuscation and comparison routine for dbghelp.dll.

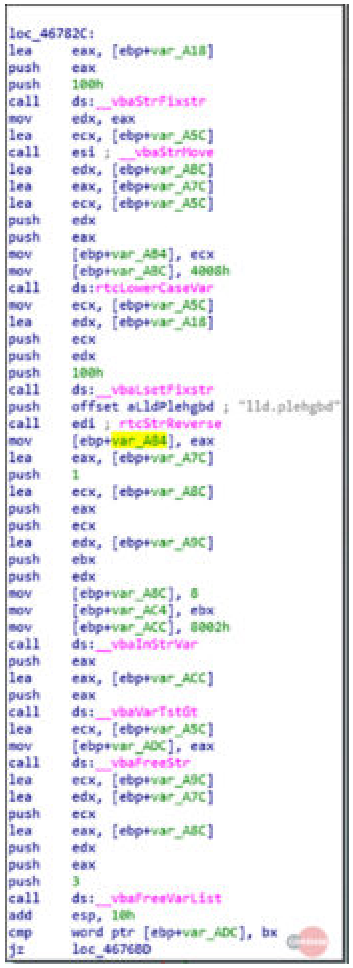
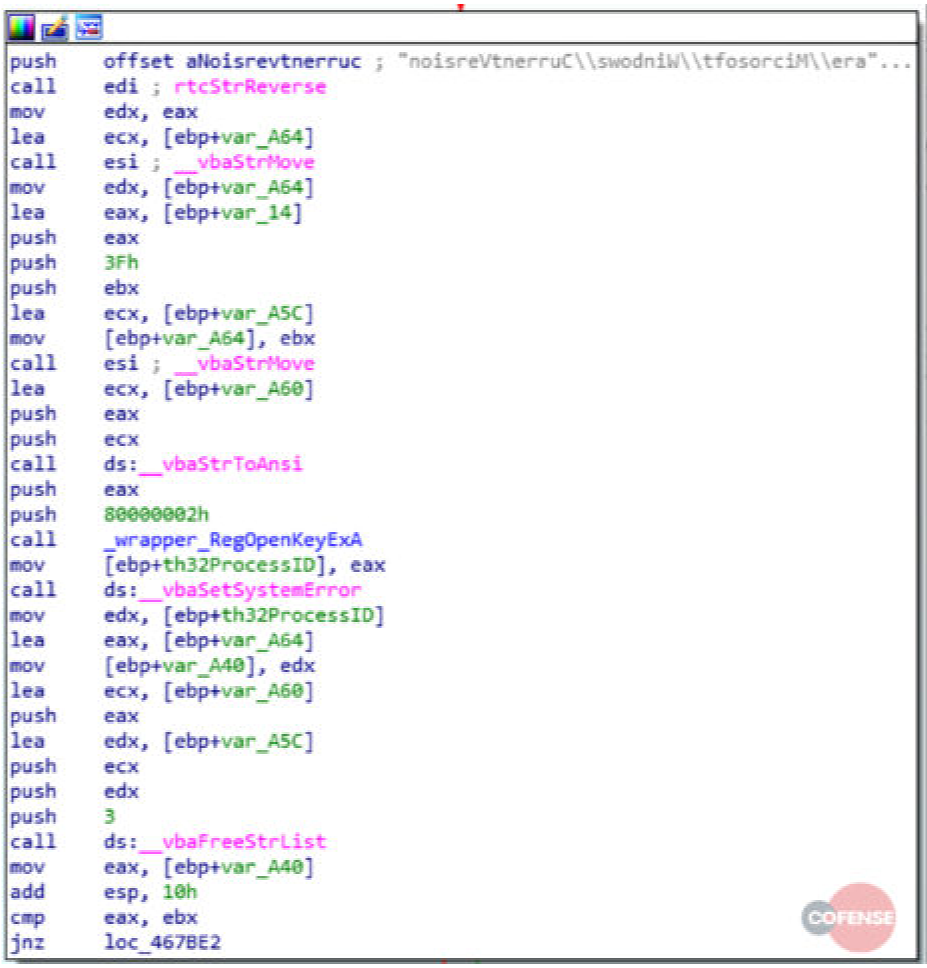


Figure 16: Windows Debug DLL de-obfuscation and string comparison.

All the comparison results are stored in a data structure, which is later checked by the _check_anti_analysis routine.

In a final check to ensure it is not being observed, Kutaki once again reads a registry key — HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion — this time checking for the existence of some very specific ProductID values within the CurrentVersion hive. Figure 17 shows this function.



```
push    offset aNoisrevtnerruc ; "noisreVtnerruC\\swodniW\\tfosorciM\\era"...
call    edi ; rtcStrReverse
mov     edx, eax
lea     ecx, [ebp+var_A64]
call    esi ; __vbaStrMove
mov     edx, [ebp+var_A64]
lea     eax, [ebp+var_14]
push    eax
push    3Fh
push    ebx
lea     ecx, [ebp+var_A5C]
mov     [ebp+var_A64], ebx
call    esi ; __vbaStrMove
lea     ecx, [ebp+var_A60]
push    eax
push    ecx
call    ds:__vbaStrToAnsi
push    eax
push    80000002h
call    _wrapper_RegOpenKeyExA
mov     [ebp+th32ProcessID], eax
call    ds:__vbaSetSystemError
mov     edx, [ebp+th32ProcessID]
lea     eax, [ebp+var_A64]
mov     [ebp+var_A40], edx
lea     ecx, [ebp+var_A60]
push    eax
lea     edx, [ebp+var_A5C]
push    ecx
push    edx
push    3
call    ds:__vbaFreeStrList
mov     eax, [ebp+var_A40]
add     esp, 10h
cmp     eax, ebx
jnz     loc_467BE2
```


Figure 17: Opening the registry key to facilitate value comparisons.

Kutaki attempts to find a value with the name “ProductID”. If it finds one key with that value, it will loop through three string comparisons, attempting to identify three sandbox platforms. Some pseudocode to roughly describe this process could be:

p_id = RegQueryValueExA(“ProductID”)

if (p_id)){

if (p_id == ‘76487-337-8429955-22614’) {

return “Anubis”

}

elif (p_id == ‘76487-644-3177037-23510’) {

return “CWSandbox”

}

elif (p_id == ‘55274-640-2673064-23950’) {

return “JoeSandbox”

}

else {

return None

}

}

Figure 18 shows this nested loop as it exists with Kutaki.

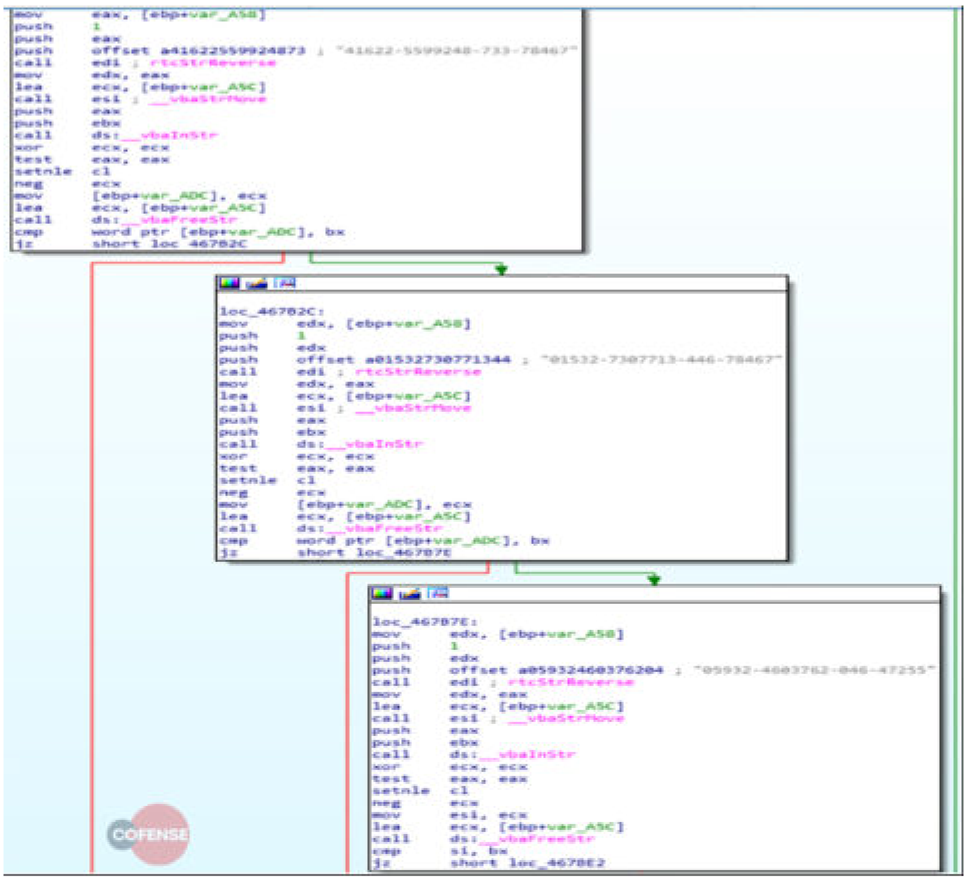


Figure 18: Looping through checks for various sandboxes.

Once all of the checks have been processed, Kutaki parses the results to determine if its main loop should finish or continue executing. The check procedure parses every result for a non-zero “return code” (i.e. something was detected) and, if it such a return code is found, the main loop exits. Figure 19 shows an example of these checks.

Figure 19: Parsing the results of one of the anti-analysis checks.

Behavior

Once Kutaki has determined it is not being monitored, it will proceed with its primary purpose of preparing the machine for data-theft. During this process, Kutaki extracts an image from its resources, drops it to the user's temp directory and launches it with ShellExecuteA("cmd.exe /c C:\Users\admin\AppData\Local\Temp\images1.png"). This displays a decoy image to fool the user into believing the OLE package they clicked on was simply an image. Figure 20 is the precise image dropped to disk and displayed to the user.


Invoice

YOUR BUSINESS NAME

CUSTOMER BUSINESS NAME

CUSTOMER BUSINESS NAME

Figure 20: Decoy image launched by Kutaki.

The document is an invoice template; clearly the actors deploying this put very little effort into this decoy document, as a quick Google search shows it to be the second image hit using the search term “tax details to invoice”. Indeed, this is the exact image used by the attackers: hxxp://batayneh[.]me/invoice-with-bank-details-template/invoice-with-bank-details-template-blank-tax-luxury/

After displaying the document, Kutaki checks its current executable name against the hardcoded string “hyuder” and, if it does not match, proceeds to drop a copy of itself, with the new name, to

C:\Users\<username>\AppData\Roaming\Microsoft\Windows\Start Menu\Programs\Startup\hyuder.exe

Figure 21 shows this check in a debugger. If a new process is launched, the parent process will sit idle without exiting.

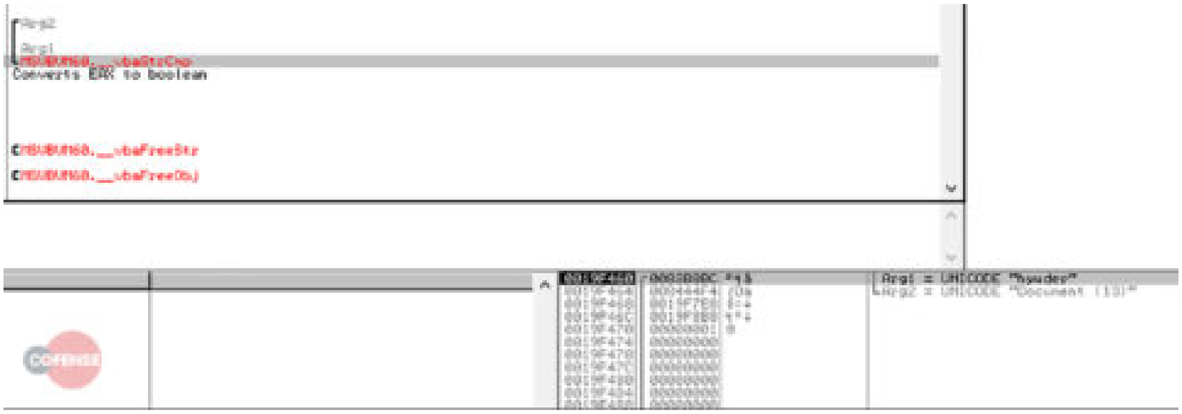


Figure 21: Kutaki comparing its current name to the desired.

Figure 22 shows Kutaki building the file path to which it will drop a copy of itself. By dropping to the startup folder, Kutaki achieves persistence.

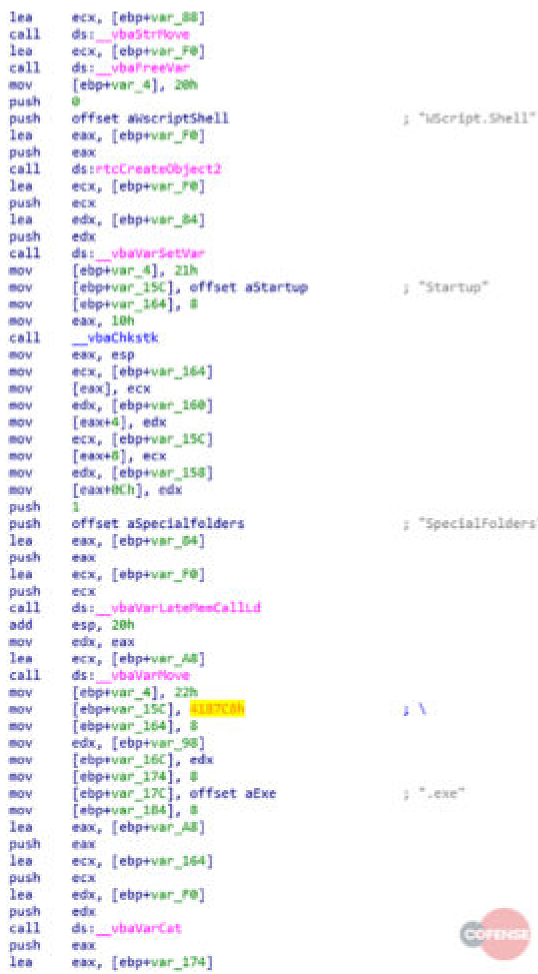


Figure 22: Kutaki builds out the string it will use to drop a copy of itself and achieve persistence.

Kutaki executes the dropped code and proceeds to execute its primary malicious functionality. Note: some readers will no doubt question whether simply renaming the executable “hyuder.exe” will prevent it from dropping a copy of itself. This is correct; Kutaki will execute directly without dropping anything further, if it is running with an approved name. The rest of the code is somewhat uninteresting, mostly because almost all the malicious behavior occurs outside of the binary itself.

Before proceeding further, Kutaki will check in with its C2 server, announcing the new infection. Figure 22 is some example traffic observed during analysis.

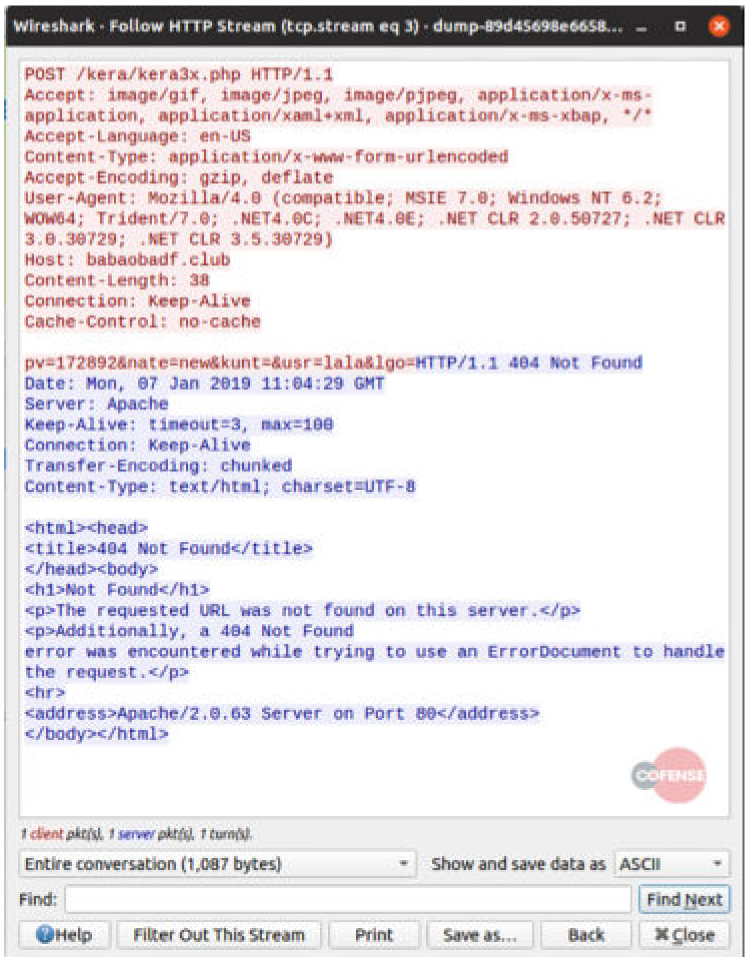


Figure 23: Kutaki’s C2 server is offline.

Kutaki also comes bundled with a copy of cURL – a Linux app ported to Windows which allows command line access to internet resources. It uses cURL to retrieve a payload from a remote host, although quite why it does this is unclear – it has the capability to contact remote servers as demonstrated by its initial attempt to contact its C2 server. Regardless, the use of cURL is by design, as the host from which it attempts to download a further payload refuses connections unless the User Agent is set to one referencing cURL. Figure 23 documents the connections made by cURL to the remote host, retrieving a secondary payload.

Figure 24: Kutaki uses cURL to download and execute a secondary payload. Note the User-Agent string “curl/7.47.1”.

The payload retrieved, in this case, was a copy of SecurityXploded’s BrowserPasswordDump. This utility is designed to retrieve passwords from the vaults of the following browsers:

- Firefox
- Google Chrome
- Microsoft Edge
- Internet Explorer
- UC Browser
- Torch Browser
- Chrome Canary/SXS Cool
- Novo Browser
- Opera Browser

- Apple Safari

Because the C2 server was offline, we were unable to monitor the exfiltration of stolen data facilitated by the BrowserPasswordDump utility.

Old Does Not Mean Ineffective

Kutaki uses some old-school, well-documented techniques to detect sandboxes and debugging. These are still effective against unhardened virtual machines and other analysis devices. Additionally, by backdooring a legitimate application, unsophisticated detection methodologies could well be fooled.

To learn more about recent malware trends, read our 2018 year-end review.

Appendix

Sources

https://www.alienvault.com/blogs/labs-research/your-malware-shall-not-fool-us-with-those-anti-analysis-tricks

https://www.fireeye.com/blog/threat-research/2011/01/the-dead-giveaways-of-vm-aware-malware.html

ProductID Checks

76487-337-8429955-22614 // Anubis Sandbox

76487-644-3177037-23510 // CW Sandbox

55274-640-2673064-23950 // Joe Sandbox

AntiVM Strings

VIRTUAL

VBOX

*VMW

sbiedll.dll

Dbghelp.dll

IoCs

hxxp://babaobadf[.]club/kera/kera3x[.]php hxxp://janawe[.]bid/FF/om2[.]exe

Artefacts

89D45698E66587279460F77BA19AE456

A69A799E2773F6D9D24D0ECF58DBD9E3

70bf5dd41548e37550882eba858c84fa

8e4aa7c4adec20a48fe4127f3cf2656d

Don't miss out on any of our phishing updates! Subscribe to our blog.