securitykitten.github.io/2017-02-15-the-rambo-backdoor.md at master · malware-kitten/securitykitten.github.io · GitHub

github.com/malware-kitten/securitykitten.github.io/blob/master/_posts/2017-02-15-the-rambo-backdoor.md

malware-kitten

malware-kitten/
securitykitten.github.io

Jekyll theme inspired by Swiss design

0 Contributors 0 Issues 0 Stars 0 Forks

Cannot retrieve contributors at this time

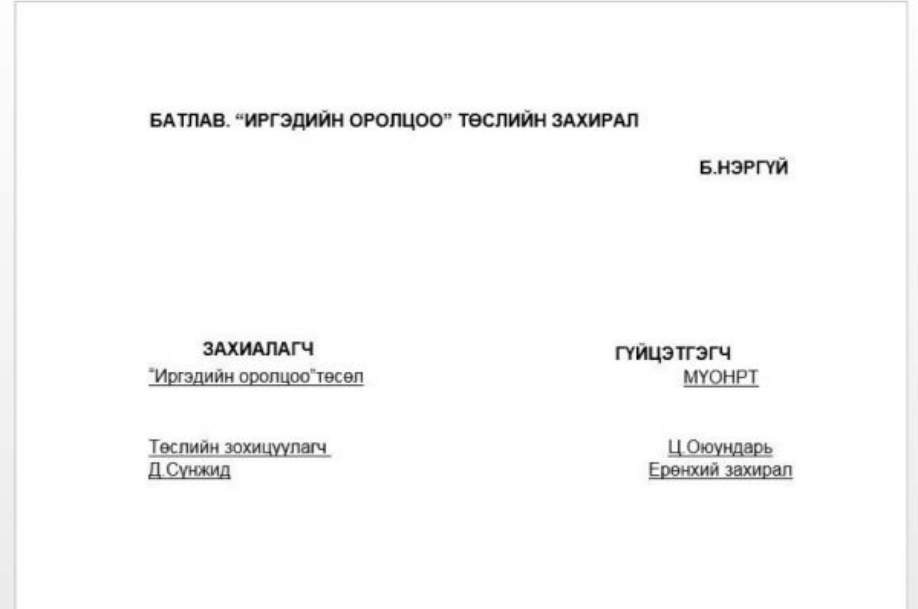
layout	title	date
category-post	The Rambo Backdoor	2017-02-15 00:00:00 -0500

Summary:

Recent new reporting was released on the DragonOK group which unveiled the many versions of the Sysget backdoor as well as the IsSpace backdoor. One of the samples we looked at `SHA256:e154e62c1936f62aeaf55a41a386dbc293050acec8c4616d16f75395884c9090` contained a family of backdoors that hasn't been referenced in public documents. In this post, we will be pulling apart and dissecting the Rambo backdoor and discussing several of its evasion techniques. This backdoor has several aliases in the community; Sophos calls the embedded components "Brebsd-A" and several others reference the code as simply "Rambo".

RTF Dropper

The initial dropper for this malware is a malicious RTF file containing many unique shellcode techniques.



БАТЛАВ. “ИРГЭДИЙН ОРОЛЦОО” ТӨСЛИЙН ЗАХИРАЛ

Б.НЭРГҮЙ

ЗАХИАЛАГЧ
“Иргэдийн оролцоо”төсөл

Төслийн зохицуулагч
Д.Сүнжид

ГҮЙЦЭТГЭГЧ
МҮОНРТ

Ц.Оюундарь
Ерөнхий захирал

Both the api hashing (ROR 7) and the save path section of code are identical. The code is also using the same payload marker of 0xbabababa.

Shellcode hashing routine

```
loc_15A:                                ; CODE XREF: seg000:00000167↓j
                movsx   edx, byte ptr [eax]
                cmp     dl, dh
                jz      short loc_169
                ror     ebx, 7
                add     ebx, edx
                inc     eax
                jmp     short loc_15A
```

The save path shellcode that is also unique to the weaponizer used in previous blogs:

```
                                ; CODE XREF: seg000:00000217↓j
inc     eax
cmp     byte ptr [ebx+eax], 0
jnz     short loc_212
mov     dword ptr [ebx+eax], '\\..'
mov     dword ptr [ebx+eax+4], 'xe..'
mov     dword ptr [ebx+eax+8], 'e'
```

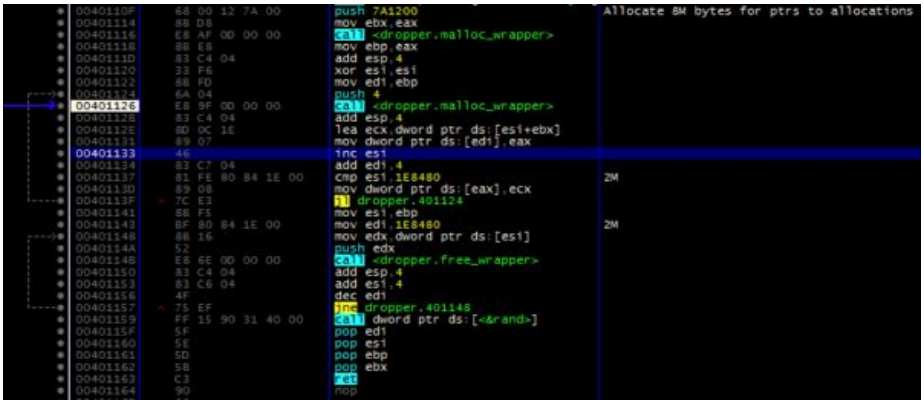
And the payload marker searching:

```
                                ; CODE XREF: seg000:00000261↓j
                                ; seg000:0000026C↓j
add     ecx, 1000h
cmp     dword ptr [edx+ecx], 0BABABABAh
jnz     short loc_254
add     edx, 4
cmp     word ptr [edx+ecx], 0BABAh
jnz     short loc_254
lea     edx, [edx+ecx+2]
xor     ebx, ebx
lea     ecx, [edi+3000h]
```

Without diving into all the intricacies of how this shellcode works it will eventually decode a payload and exec it. The parser that PAN provided will also work when extracting the payload from this document.

Rambo

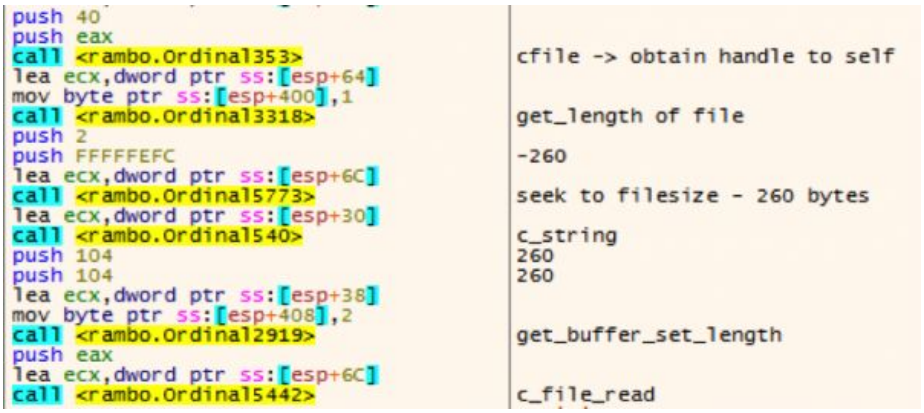
Quickly after starting up, Rambo proceeds to enter a busy-loop making 2 million small malloc calls and then freeing each allocation. This ties up the malware for a couple minutes in order to throw off AV emulators (which will only emulate so many instructions). This also helps evade most sandboxes. Now that many sandbox systems short-circuit the sleep call, more malware is moving from sleeping to busy loops in order to use up the short time slice that a sandbox can devote to each sample.



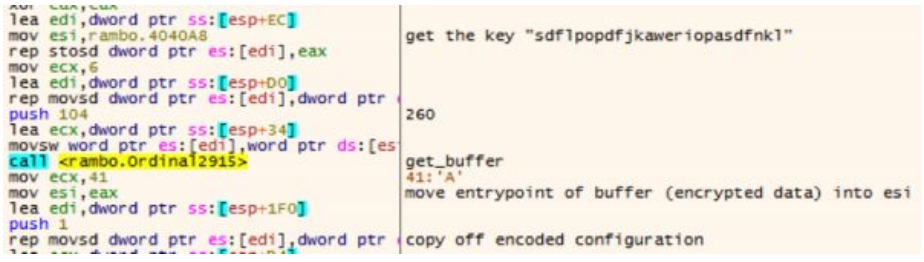
Rambo contains several different components working in tandem to achieve full execution on the victim machine. The initial binary SHA256: `7571642ec340c4833950bb86d3ded4d4b7c2068347e8125a072c5a062a5d6b68` is a dropper that unpacks the 3 different parts, achieves persistence and starts execution. The dropper is also copied as the method of persistence.

The key `HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Run\FaultCheck` is established at the persistence key with the key value pointing at `C:\Users\<username>\AppData\Local\Temp\<filename>`

Rambo will then fetch its configuration by reading in the last 260 bytes of itself.



The key “sdflpopdfjkaweriopasdfnkl” is loaded, which is eventually used to decrypt the buffer using tiny encryption algorithm (TEA).



Even though the whole string is referenced as a string, only the first 16 characters are used as the functional key. Perhaps this is a misunderstanding of the author, or an attempt to throw off analysts. The steps of the TEA decryption can be seen below.

```
mov ecx,C6EF3720                        sum constant
jmp rambo.401D4A
cmp edx,10
jne rambo.401D42
mov ecx,E3779B90
jmp rambo.401D4A
mov ecx,edx
imul ecx,ecx,9E3779B9                   delta
mov eax,edx
dec edx
test eax,eax
je rambo.401DA3
inc edx
mov ebx,dword ptr ss:[esp+24]
mov ebp,dword ptr ss:[esp+10]
mov eax,edi
shr eax,5                               shift right
add eax,ebx
mov ebx,edi
shl ebx,4                               shift left
add ebx,ebp
mov ebp,dword ptr ss:[esp+18]
xor eax,ebx
lea ebx,dword ptr ds:[ecx+edi]
xor eax,ebx
mov ebx,dword ptr ss:[esp+14]
sub esi,eax
mov eax,esi
shr eax,5                               shift right
add eax,ebx
mov ebx,esi
shl ebx,4                               shift left
add ebx,ebp
xor eax,ebx
lea ebx,dword ptr ds:[ecx+esi]
xor eax,ebx
add ecx,61C88647                        add rather than subtract
sub edi,eax
dec edx
```

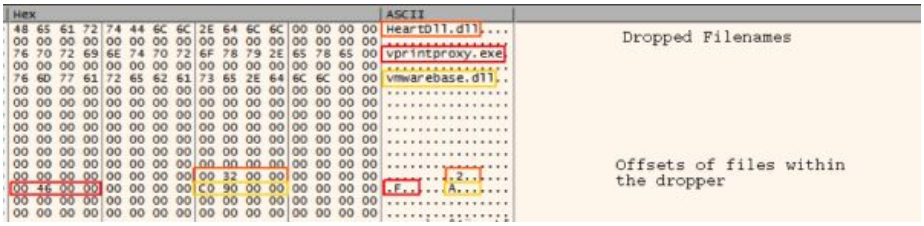
The decryption of the code can be translated to python with the following snippet. (To get the decryption working, we had to make some patches to the opensource PyTea implementation, a modified copy of the script that is used is posted at the end of this blogpost)

```
#!/usr/bin/env python
from ctypes import *
from pprint import pprint
import sys
import tea
import re
import struct

def ascii_strings(data):
    strings = []
    for match in re.finditer(r'[\x20-\x80\n\r\t]{16,64}',data):
        strings.append(match.group()[:16])
    return strings

def to_c_array(data):
    ''' Converts a string to a list of c_uint32s '''
    c_array = []
    char_array = [hex(ord(char))[2:] for char in data]
    for index in range(0, len(char_array), 4):
        block = char_array[index:index + 4]
        hex_value = '0x' + ''.join(block)
        c_array.append(c_uint32(int(hex_value, 16)))
    return c_array

with open(sys.argv[1], 'rb') as fp:
    data = fp.read()

ciphertext = data[-260:]
padding = len(ciphertext)%8
ciphertext += '\x00'*padding

for key in ascii_strings(data):
    #print 'trying key %s' % (key)
    try:
        plaintext = tea.decrypt(ciphertext, key,verbose=False)
        if ".dll" in plaintext.lower() or '.exe' in plaintext.lower():
            break
    except:
        pass

plaintext = plaintext[:-padding]
print '[*]\tDecrypted with key "%s"\nConfig:' % (key)
config = {}
config['loader'] = {'name': plaintext[:0x20].rstrip('\x00'),
                    'offset': struct.unpack('<L', plaintext[0xc8:0xcc])[0]}
config['sideloader'] = {'name': plaintext[0x20:0x40].rstrip('\x00'),
                    'offset': struct.unpack('<L', plaintext[0xd0:0xd4])[0]}
config['backdoor'] = {'name': plaintext[0x40:0x60].rstrip('\x00'),
                    'offset': struct.unpack('<L', plaintext[0xd8:0xdc])[0]}
config['loader']['length'] = config['sideloader']['offset'] - config['loader']['offset']
config['sideloader']['length'] = config['backdoor']['offset'] - config['sideloader']['offset']
config['backdoor']['length'] = len(data) - config['backdoor']['offset'] - 260
pprint(config)
print

for key, component in config.items():
    with open(component['name'], 'wb') as fp:
        print '[*]\tDropping %s' % (component['name'])
        fp.write(data[component['offset']:component['offset']+component['length']])
```

Running the above script will yield in the following information and drop the 3 components:

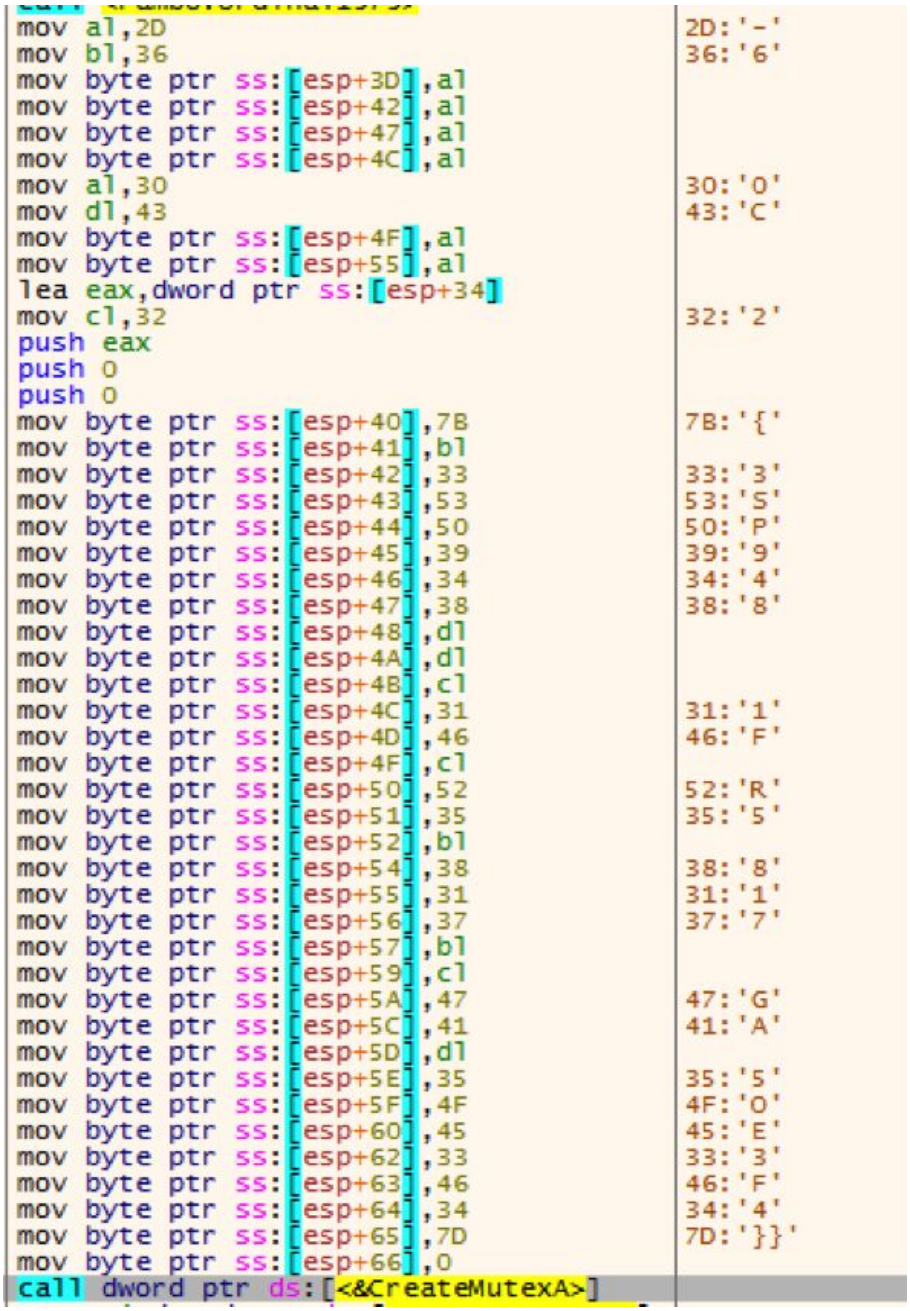
```
[*]    Decrypted with key "sdflpopdfjkaweri"
Config:
{'backdoor': {'length': 14336, 'name': 'vmwarebase.dll', 'offset': 37056},
 'loader': {'length': 5120, 'name': 'HeartDll.dll', 'offset': 12800},
 'sideloader': {'length': 19136, 'name': 'vprintproxy.exe', 'offset': 17920}}
[*]    Dropping vmwarebase.dll
[*]    Dropping vprintproxy.exe
[*]    Dropping HeartDll.dll
```

The configuration contains the names of the dropped files and the offsets of each file. Marked up, the configuration will resemble the following.



Once the configuration is decoded the malware will carve each file out and write them to disk.

Rambo (and the embedded components) make heavy use of stack strings to evade basic triage (ie, strings) from revealing a lot of information.



The mutex is created with the value of `{63SP948C-C21F-2R56-8176-2G0AC5OE03F4}`. Once the mutex is created, WinExec is called starting HeartDll.dll with the DllRegisterServer argument.

```
"cmd /c rundll32.exe \"C:\\Users\\admin\\AppData\\Local\\Temp\\HeartDll.dll\",DllRegisterServer"
```

HeartDll.dll

HeartDll.dll (SHA256: `11668a0666636b3c40b61986bf132a8ca6ab448fddcaa9e4ed22f6ca7f7b8a50`) is a small executable (roughly 5k in size). This is responsible to starting vprintproxy (which ultimately sideloads vmwarebase.dll).

Upon initial execution, HeartDll.dll will create a mutex (statically configured) of `{53A7Y6CC-C8EF-W089-CN21-220AQCD303F3}`

At the startup of HeartDll.dll it'll load 4 different commands into a buffer.

- bsd -1

- bre -1
- esd +2
- ere +2

HeartDll.dll will write “bsd -1” to file 1.txt which will seed a command for the backdoor when it starts executing.

First the dll will locate the current working directory and manually build the string “vprintproxy.exe”

```
mov     [ebp+Source], 'v'
mov     [ebp+var_23], 'p'
mov     [ebp+var_22], 'r'
mov     [ebp+var_21], 'i'
mov     [ebp+var_20], 'n'
mov     [ebp+var_1F], 't'
mov     [ebp+var_1E], 'p'
mov     [ebp+var_1D], 'r'
mov     [ebp+var_1C], 'o'
mov     [ebp+var_1B], 'x'
mov     [ebp+var_1A], 'y'
mov     [ebp+var_19], '.'
mov     [ebp+var_18], 'e'
mov     [ebp+var_17], 'x'
mov     [ebp+var_16], 'e'
mov     [ebp+var_15], bl
call    strcat
```

Heart will write the contents of 1.txt into a file named 222.txt. Once this is done then heart will call WinExec on vprintproxy.exe which will in turn sideload the malicious vmwarebase.dll.

At this point, it’ll enter an infinite loop of sleeping and attempting to read the file 3.txt. Which contains startup information from vmwarebase.dll. It’ll loop through the various expect log messages and then exit.

vprintproxy.exe

This is legit executable that is signed by VMWare that the authors use to sideload vmwarebase.dll

Copyright	Copyright © 1998-2015 VMware, Inc.
Product	VMware Workstation
Original name	vprintproxy.exe
Internal name	vprintproxy
File version	11.1.2 build-2780323
Description	VMware VPrint Proxy
Signature verification	Signed file, verified signature
Signing date	3:09 PM 5/31/2015
Signers	[+] VMware [+] VeriSign Class 3 Code Signing 2010 CA [+] VeriSign
Counter signers	[+] Symantec Time Stamping Services Signer - G4 [+] Symantec Time Stamping Services CA - G2 [+] Thawte Timestamping CA

The imports directly load vmwarebase.dll

004020CC	ProductState_GetCompilationOption	vmwarebase
004020D0	Log	vmwarebase
004020D4	Win32U_RegCreateKeyEx	vmwarebase
004020D8	Win32U_RegOpenKeyEx	vmwarebase
004020DC	Err_Errno2String	vmwarebase
004020E0	Win32U_LoadLibrary	vmwarebase
004020E4	W32Util_GetInstalledFilePath	vmwarebase
004020E8	Warning	vmwarebase
004020EC	W32Util_AsciiStrToWideStr	vmwarebase
004020F0	Preference_Init	vmwarebase
004020F4	ProductState_GetBuildNumberString	vmwarebase
004020F8	ProductState_GetVersion	vmwarebase
004020FC	ProductState_GetName	vmwarebase
00402100	Log_SetProductInfo	vmwarebase
00402104	Log_CfgInterface	vmwarebase
00402108	Log_InitWithFileSimpleInt	vmwarebase

vmwarebase.dll

Vmwarebase.dll is loaded up via vprintproxy.exe and contains much of the functionality of this family.

When loading up, it'll decode its configuration via a simple xor loop.

```
jle vmwarebase.10001450
mov dl,byte ptr ss:[esp+8]
xor byte ptr ds:[eax+ecx],dl
inc eax
cmp eax,dword ptr ss:[esp+C]
jl vmwarebase.10001442
ret
```

In this case the decoded c2 is busserh.mancely.com.

```
EAX   00000013
EBX   00000000
ECX   0034F278     "busserh.mancely.com"
EDX   7F15137B
EBP   0034F36C
ESP   0034F03C
ESI   00000001
EDI   0000007B     '{'

EIP   10001450     vmwarebase.10001450
```

During its execution, the malware will use the same loop to decode its port information (443 & 80) and other configuration information.

Once the configuration information is parsed, the malware will load up the same debug messages as HeartDll.dll (bre -1, bsd -1, ere +2, and esd +2), these are used primary as communication between HeartDll.dll

It'll attempt to read 1.txt, and if the information in 1.txt matches "bsd -1", the malware will recon information off the host and send it to the c2 controller.

Host Recon

In the main reconnaissance function, the malware will grab the system proxy settings from the registry key "Software\Microsoft\Windows\CurrentVersion\Internet Settings\ProxyServer". By pulling this information, this may ensure a slightly higher success rate of communicating out in a corporate environment. As the case with all these binaries, it makes heavy use of manually building stack strings to evade the simple strings tool.

```
mov     [ebp+var_26], 'I'
mov     [ebp+var_25], 'n'
mov     [ebp+var_24], 't'
mov     [ebp+var_23], 'e'
mov     [ebp+var_22], 'r'
mov     [ebp+var_21], 'n'
mov     [ebp+var_20], 'e'
mov     [ebp+var_1F], 't'
mov     [ebp+var_1E], ' '
mov     [ebp+var_1D], 'S'
mov     [ebp+var_1C], 'e'
mov     [ebp+var_1B], 't'
mov     [ebp+var_1A], 't'
mov     [ebp+var_19], 'i'
mov     [ebp+var_18], 'n'
mov     [ebp+var_17], 'g'
mov     [ebp+var_16], 's'
mov     [ebp+var_15], bl
push    eax             ; lpSubKey
push    80000001h       ; hKey
call    ds:RegOpenKeyExA
```

Rambo will continue to gather the hostname and IP of the system. Gather a list of processes (with a PID of greater than 10) by calling CreateToolhelp32Snapshot. It'll also grab the Windows version and CPU arch.

Prior to encryption, the contents of the buffer before it's sent out to the C2 contains the following information:

```
10.152.X.X|##HOSTNAME##d##OPOP<*
<smss.exe>>csrss.exe>>wininit.exe>>csrss.exe>>winlogon.exe>>services.exe>>lsass.exe>>lsm.exe>>svch
 >>taskmgr.exe>>notepad.exe>>cmd.
exe>>conhost.exe>>rundll32.exe>>cmd.exe>>conhost.exe>>SearchProtocolHost.exe>>Search
FilterHost.exe>>conhost.exe>><*<6.1.7601 Service Pack 1>>x64>>409>>
```

C2 communications

The data that is harvested from the host is sent to the C2 controller and encrypted using an AES key of \x12\x44\x56\x38\x55\x82\x56\x85\x23\x25\x56\x45\x52\x47\x45\x86. In ascii, (while not all characters are printable), the string will be "\x12DV8U\x82V\x85#%VERGE\x86".

Once the function is finished, it'll write "esd +2" to the file 222.txt.

Download and Execute

If the file 1.txt contains the command "bre -1" the malware will continue down a different path of execution.

The malware will generate a random filename (8 characters long), by using a lookup table. It'll generate indexes into the string "123456789abcdefehijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ" and simply concat them together.

The proxy settings are read again and a simple connect is performed. If the connect succeeds "ok" is sent.

The recv call is performed and a file is downloaded, written to the temporary file name and exec'd using the following hardcoded command.

```
cmd.exe /c rundll32.exe <filename>,FSSync_ScreeActive
```

```
mov     [ebp+var_47], 'm'
mov     [ebp+var_46], 'd'
mov     [ebp+var_45], ' '
mov     [ebp+var_44], '/'
mov     [ebp+var_43], 'c'
mov     [ebp+var_42], ' '
mov     [ebp+var_41], 'r'
mov     [ebp+var_40], 'u'
mov     [ebp+var_3F], 'n'
mov     [ebp+var_3E], 'd'
mov     [ebp+var_3D], 'l'
mov     [ebp+var_3C], 'l'
mov     [ebp+var_3B], '3'
mov     [ebp+var_3A], '2'
mov     [ebp+var_39], '.'
mov     [ebp+var_38], 'e'
mov     [ebp+var_37], 'x'
mov     [ebp+var_36], 'e'
mov     [ebp+var_35], ' '
mov     [ebp+var_34], '"'
mov     [ebp+var_33], bl
call    strcpy
push    [ebp+Filename]  ; Source
lea     eax, [ebp+CmdLine]
push    eax             ; Dest
call    strcat
lea     eax, [ebp+var_30]
mov     [ebp+var_30], '"'
push    eax             ; Source
lea     eax, [ebp+CmdLine]
push    eax             ; Dest
mov     [ebp+var_2F], ','
mov     [ebp+var_2E], 'F'
mov     [ebp+var_2D], 'S'
mov     [ebp+var_2C], 'S'
mov     [ebp+var_2B], 'y'
mov     [ebp+var_2A], 'n'
mov     [ebp+var_29], 'c'
mov     [ebp+var_28], '_'
mov     [ebp+var_27], 'S'
mov     [ebp+var_26], 'c'
mov     [ebp+var_25], 'r'
mov     [ebp+var_24], 'e'
mov     [ebp+var_23], 'e'
mov     [ebp+var_22], 'A'
mov     [ebp+var_21], 'c'
mov     [ebp+var_20], 't'
mov     [ebp+var_1F], 'i'
mov     [ebp+var_1E], 'v'
mov     [ebp+var_1D], 'e'
```

During the course of research, we didn't identify the secondary file that is pushed to the host, although some information can be gained from static analysis. The file would need to be PE DLL with an exported function of FSSync_ScreeActive. This is most likely the function in which the authors will load a more robust stage 2 backdoor.

When the command is completed, "ere +2" is written to 222.txt

Summary

Rambo is a unique backdoor with features that are the result of some odd design decisions. In the initial dropper the configuration containing offsets and filenames are encoded with TEA, however the binaries are not encoded at all. It uses AES to encode the host information that is sent out over the network, however the C2 is hidden with a single byte XOR. While they may not make much sense to a reverse engineer, it gives some idea to the information that the author doesn't want to be easily recovered. By writing commands to temporary files and trying to communicate between multiple processes, the authors turn a simple stage 1 implant into something that is confusing and more difficult to study.

Mature security programs research edge cases and newly discovered code in order to understand tools, tactics and procedures of successful advanced groups that will inevitably become more common in the future.

Indicators of Compromise:

Indicator	Type	Description
busserh.mancely.com	Domain	Command and Control
gosuper@excite.co.jp	Email Address	Registrant of busserh.mancely.com
108.61.117.31	IP	Resolution of busserh.mancely.com
C:\Users<user>\AppData\Local\Temp\HeartDll.dll	Filename	
C:\Users<user>\AppData\Local\Temp\vprintproxy.exe	Filename	
C:\Users<user>\AppData\Local\Temp\vmwarebase.dll	Filename	
C:\Users<user>\AppData\Local\Temp\222.txt	Filename	
C:\Users<user>\AppData\Local\Temp\3.txt	Filename	
e154e62c1936f62aeaf55a41a386dbc293050acec8c4616d16f75395884c9090	Hash	RTF Dropper
7571642ec340c4833950bb86d3ded4d4b7c2068347e8125a072c5a062a5d6b68	Hash	Main Dropper
5bfcd2cc01a5b930fc704a695f0fe38f1bca8bdfafd8b7d931a37428b5e86f35	Hash	Hash of vmwarebase.dll
76405617acc7fa6c51882fe49d9b059900c10fc077840df9f6a604bf4fab85ba	Hash	Hash of vprintproxy.exe (legit executable)
11668a0666636b3c40b61986bf132a8ca6ab448fddcaa9e4ed22f6ca7f7b8a50	Hash	Hash of HeartDll.dll

Additional Notes

In the symbol table for Rambo (vmwarebase.dll) it appears that the authors left in the original compiled name of the executable (FirstBlood.tmp) which accounts for the naming convention.

```
Export directory for FirstBlood.tmp

            dd 0                    ; Characteristics
            dd 57FC3359h            ; TimeDateStamp: Tue Oct 11 00:33:29 2016
            dw 0                    ; MajorVersion
            dw 0                    ; MinorVersion
            dd rva aFirstblood_tmp  ; Name
            dd 1                    ; Base
            dd 10h                  ; NumberOfFunctions
            dd 10h                  ; NumberOfNames
            dd rva off_10004658     ; AddressOfFunctions
            dd rva off_10004698     ; AddressOfNames
            dd rva word_100046D8    ; AddressOfNameOrdinals
```

The functions that contain the name are the functions that were overwritten from the legit vmwarebase.dll as to not break the functionality of vprintproxy.exe.

```
vaddr=0x10001431 paddr=0x00000831 ord=000 fwd=NONE sz=0 bind=GLOBAL type=FUNC
name=FirstBlood.tmp_Err_Errno2String
vaddr=0x10001431 paddr=0x00000831 ord=001 fwd=NONE sz=0 bind=GLOBAL type=FUNC
name=FirstBlood.tmp_Log
vaddr=0x10001431 paddr=0x00000831 ord=002 fwd=NONE sz=0 bind=GLOBAL type=FUNC
name=FirstBlood.tmp_Log_CfgInterface
vaddr=0x10001431 paddr=0x00000831 ord=003 fwd=NONE sz=0 bind=GLOBAL type=FUNC
name=FirstBlood.tmp_Log_InitWithFileSimpleInt
vaddr=0x10001431 paddr=0x00000831 ord=004 fwd=NONE sz=0 bind=GLOBAL type=FUNC
name=FirstBlood.tmp_Log_SetProductInfo
vaddr=0x10001431 paddr=0x00000831 ord=005 fwd=NONE sz=0 bind=GLOBAL type=FUNC
name=FirstBlood.tmp_Preference_Init
vaddr=0x10001431 paddr=0x00000831 ord=006 fwd=NONE sz=0 bind=GLOBAL type=FUNC
name=FirstBlood.tmp_ProductState_GetBuildNumberString
vaddr=0x10001431 paddr=0x00000831 ord=007 fwd=NONE sz=0 bind=GLOBAL type=FUNC
name=FirstBlood.tmp_ProductState_GetCompilationOption
vaddr=0x10001431 paddr=0x00000831 ord=008 fwd=NONE sz=0 bind=GLOBAL type=FUNC
name=FirstBlood.tmp_ProductState_GetName
vaddr=0x10001431 paddr=0x00000831 ord=009 fwd=NONE sz=0 bind=GLOBAL type=FUNC
name=FirstBlood.tmp_ProductState_GetVersion
vaddr=0x10001431 paddr=0x00000831 ord=010 fwd=NONE sz=0 bind=GLOBAL type=FUNC
name=FirstBlood.tmp_W32Util_AsciiStrToWideStr
vaddr=0x10001431 paddr=0x00000831 ord=011 fwd=NONE sz=0 bind=GLOBAL type=FUNC
name=FirstBlood.tmp_W32Util_GetInstalledFilePath
vaddr=0x10001431 paddr=0x00000831 ord=012 fwd=NONE sz=0 bind=GLOBAL type=FUNC
name=FirstBlood.tmp_Warning
vaddr=0x10001431 paddr=0x00000831 ord=013 fwd=NONE sz=0 bind=GLOBAL type=FUNC
name=FirstBlood.tmp_Win32U_LoadLibrary
vaddr=0x10001431 paddr=0x00000831 ord=014 fwd=NONE sz=0 bind=GLOBAL type=FUNC
name=FirstBlood.tmp_Win32U_RegCreateKeyEx
vaddr=0x10001431 paddr=0x00000831 ord=015 fwd=NONE sz=0 bind=GLOBAL type=FUNC
name=FirstBlood.tmp_Win32U_RegOpenKeyEx
```

Modified PyTEA

```
#!/usr/bin/env python

################################################################################
# Python implementation of the Tiny Encryption Algorithm (TEA)
# By Moloch
#
# About: TEA has a few weaknesses. Most notably, it suffers from
#        equivalent keys each key is equivalent to three others,
#        which means that the effective key size is only 126 bits.
#        As a result, TEA is especially bad as a cryptographic hash
#        function. This weakness led to a method for hacking Microsoft's
#        Xbox game console (where I first encountered it), where the
#        cipher was used as a hash function. TEA is also susceptible
#        to a related-key attack which requires 2^23 chosen plaintexts
#        under a related-key pair, with 2^32 time complexity.
#
#        Block size: 64bits
#          Key size: 128bits
#
################################################################################

import os
import getpass
import platform
import struct
from random import choice
from hashlib import sha256
from ctypes import c_uint32
from string import ascii_letters, digits

if platform.system().lower() in ['linux', 'darwin']:
    INFO = "\033[1m\033[36m[*]\033[0m "
    WARN = "\033[1m\033[31m[!]\033[0m "
else:
    INFO = "[*] "
    WARN = "[!] "


### Magical Constants
DELTA = 0x9e3779b9
SUMATION = 0xc6ef3720
ROUNDS = 32
BLOCK_SIZE = 2  # number of 32-bit ints
KEY_SIZE = 4


### Functions ###
def encrypt_block(block, key, verbose=False):
    '''
    Encrypt a single 64-bit block using a given key
    @param block: list of two c_uint32s
    @param key: list of four c_uint32s
    '''
    assert len(block) == BLOCK_SIZE
    assert len(key) == KEY_SIZE
    sumation = c_uint32(0)
    delta = c_uint32(DELTA)
    for index in range(0, ROUNDS):
        sumation.value += delta.value
        block[0].value += ((block[1].value << 4) + key[0].value) ^ (block[1].value +
sumation.value) ^ ((block[1].value >> 5) + key[1].value)
        block[1].value += ((block[0].value << 4) + key[2].value) ^ (block[0].value +
sumation.value) ^ ((block[0].value >> 5) + key[3].value)
        if verbose: print("\t--> Encrypting block round %d of %d" % (index + 1, ROUNDS))
    return block

def decrypt_block(block, key, verbose=False):
    '''
    Decrypt a single 64-bit block using a given key
```

```
    @param block: list of two c_uint32s
    @param key: list of four c_uint32s
    '''
    assert len(block) == BLOCK_SIZE
    assert len(key) == KEY_SIZE
    sumation = c_uint32(SUMATION)
    delta = c_uint32(DELTA)
    for index in range(0, ROUNDS):
        block[1].value -= ((block[0].value << 4) + key[2].value) ^ (block[0].value +
sumation.value) ^ ((block[0].value >> 5) + key[3].value);
        block[0].value -= ((block[1].value << 4) + key[0].value) ^ (block[1].value +
sumation.value) ^ ((block[1].value >> 5) + key[1].value);
        sumation.value -= delta.value
        if verbose: print("\t<-- Decrypting block round %d of %d" % (index + 1, ROUNDS))
    return block

def to_c_array(data):
    ''' Converts a string to a list of c_uint32s '''
    c_array = []
    for index in range(0, len(data)/4):
        chunk = data[index*4:index*4+4]
        packed = struct.unpack(">L", chunk)[0]
        c_array.append(c_uint32(packed))
    return c_array

def to_string(c_array):
    ''' Converts a list of c_uint32s to a Python (ascii) string '''
    output = ''
    for block in c_array:
        output += struct.pack(">L", block.value)
    return output

def random_chars(nchars):
    chars = ''
    for n in range(0, nchars):
        chars += choice(ascii_letters + digits)
    return chars

def add_padding(data, verbose=False):
    pad_delta = 4 - (len(data) % 4)
    if verbose:
        print(INFO + "Padding delta: %d" % pad_delta)
    data += random_chars(pad_delta)
    data += "%s%d" % (random_chars(3), pad_delta)
    return data

def encrypt(data, key, verbose=False):
    '''
    Encrypt string using TEA algorithm with a given key
    '''
    data = add_padding(data, verbose)
    data = to_c_array(data)
    key = to_c_array(key.encode('ascii', 'ignore'))
    cipher_text = []
    for index in range(0, len(data), 2):
        if verbose:
            print(INFO + "Encrypting block %d" % index)
        block = data[index:index + 2]
        block = encrypt_block(block, key, verbose)
        for uint in block:
            cipher_text.append(uint)
    if verbose:
        print(INFO + "Encryption completed successfully")
    return to_string(cipher_text)

def decrypt(data, key, verbose=False):
    data = to_c_array(data)
    key = to_c_array(key.encode('ascii', 'ignore'))
    plain_text = []
```

```
    for index in range(0, len(data), 2):
        if verbose:
            print(INFO + "Encrypting block %d" % index)
        block = data[index:index + 2]
        decrypted_block = decrypt_block(block, key, verbose)
        for uint in decrypted_block:
            plain_text.append(uint)
    data = to_string(plain_text)
    if verbose:
        print(INFO + "Decryption compelted successfully")
    return data

def get_key(password=''):
    ''' Generate a key based on user password '''
    if 0 == len(password):
        password = getpass.getpass(INFO + "Password: ")
    sha = sha256()
    sha.update(password + "Magic Static Salt")
    sha.update(sha.hexdigest())
    return ''.join([char for char in sha.hexdigest()[::4]])

def encrypt_file(fpath, key, verbose=False):
    with open(fpath, 'rb+') as fp:
        data = fp.read()
        cipher_text = encrypt(data, key, verbose)
        fp.seek(0)
        fp.write(cipher_text)
    fp.close()

def decrypt_file(fpath, key, verbose=False):
    with open(fpath, 'rb+') as fp:
        data = fp.read()
        plain_text = decrypt(data, key, verbose)
        fp.close()
    fp = open(fpath, 'w')
    fp.write(plain_text)
    fp.close()

### UI Code ###
if __name__ == '__main__':
    from argparse import ArgumentParser
    parser = ArgumentParser(
        description='Python implementation of the TEA cipher',
    )
    parser.add_argument('-e', '--encrypt',
        help='encrypt a file',
        dest='epath',
        default=None
    )
    parser.add_argument('-d', '--decrypt',
        help='decrypt a file',
        dest='dpath',
        default=None
    )
    parser.add_argument('--verbose',
        help='display verbose output',
        default=False,
        action='store_true',
        dest='verbose'
    )
    args = parser.parse_args()
    if args.epath is None and args.dpath is None:
        print('Error: Must use --encrypt or --decrypt')
    elif args.epath is not None:
        print(WARN + 'Encrypt Mode: The file will be overwritten')
        if os.path.exists(args.epath) and os.path.isfile(args.epath):
            key = get_key()
            encrypt_file(args.epath, key, args.verbose)
        else:
```

```
        print(WARN + 'Error: target does not exist, or is not a file')
elif args.dpath is not None:
    print(WARN + 'Decrypt Mode: The file will be overwritten')
    if os.path.exists(args.dpath) and os.path.isfile(args.dpath):
        key = get_key()
        decrypt_file(args.dpath, key, args.verbose)
    else:
        print(WARN + 'Error: target does not exist, or is not a file')
```