TrendLabs

New Exploit Kit Underminer Delivers Bootkit and Cryptocurrency-mining Malware with Encrypted TCP Tunnel

Technical Brief

TrendLabs Security Intelligence Blog
Jaromir Horejsi and Joseph C. Chen
Cyber Safety Solutions Team
July 2018

Underminer's Infection Chain

Shellcode in SWF exploit

The array of decimal numbers stored in the vector inside the SWF file is converted into a binary shellcode. The initial instructions of the shellcode determine the system's architecture (32-bit or 64-bit).

```
class PayloadWin32
{

   public static var payload:Vector.<uint> = Vector.<uint>([2420162609,70681615,1620836352,1768,2013270272,

   function PayloadWin32()
   {
      super();
   }
}
```

Figure 1: Vector containing a binary shellcode

The code below behaves differently. If run on a system with 32-bit architecture, a machine code (seen below) is disassembled into an assembly listing as shown in Figure 1. The EAX register value is clearly set to 1 and the conditional jump is not taken.

When run on 64-bit machine, the machine code is disassembled to the code below (Figure 2, bottom). This causes the EAX register to remain zero and the conditional jump is taken to a 64-bit code branch. In both cases, the shellcode downloads an additional stage of the payload, starts a new rundll32.exe process, then injects and executes the newly downloaded payload in the context of the newly created process.

```
          31 C0 40 90 0F 84 36 04  00 00
00000000                  xor     eax, eax
00000002                  inc     eax
00000003                  nop
00000004                  jz      loc_440

0000000000000000                  xor     eax, eax
0000000000000002                  xchg    eax, eax
0000000000000004                  jz      loc_440
```

Figure 2. The machine code (top), and the assembly listing (center), and the assembly listing in a 64-bit machine (bottom)

Stage 1: wasm downloader

This stage is downloaded from a url similar to http://<IP>:<PORT>/rt/<NAME>.wasm. Although wasm pertains to WebAssembly, it's not the case here. The wasm here is an executable file with a stripped MZ header, so simply running or debugging it is not possible unless the security analyst adds headers, resolves imports, and fixes relocations.

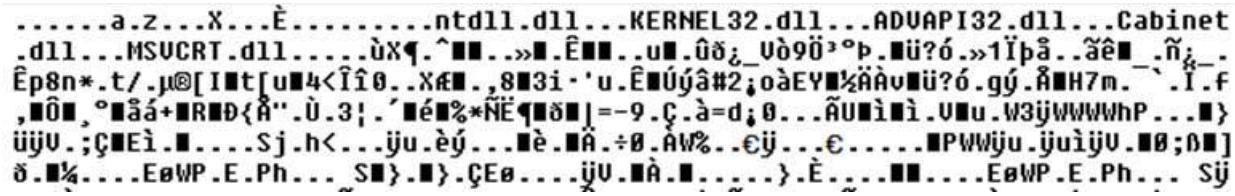

Figure 3. Beginning of stage 1; the wasm downloader is a Windows executable file but its headers are stripped

The offset of the entry point as well as imports listed in the header of the wasm file must be resolved at runtime, as shown in Figure 4.

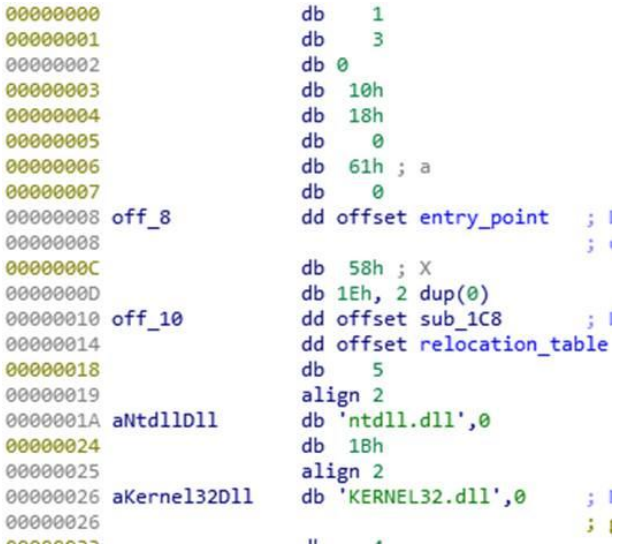

Figure 4: Custom format of the wasm binary file

The stage 1 downloader has a base64 encoded string embedded in its binary. When decoded, it reveals URLs with the following stages:

```
http://103.35.72.223:8080/git/wiki.asp?id=519358d46ad51d9628cbe9059a502b67
http://103.35.72.223:8080/git/glfw.wasm
http://103.35.72.223:8080/rt/flolroam9415u2es53ogcl76ok.wasm
```

Figure 5. URLs embedded in stage 1 after decoding

During our analysis, the *wiki.asp* webpage did not return any content. It is possibly used for reporting/statistics. The second URL highlighted in Figure 5 returned the stage 2 of the infection. Downloading the stage 2 payload shows that it is a cabinet file. Stage 1 unpacks the cabinet in memory, creates a dllhost.exe process, injects the payloads into the newly created process, then executes it. The

cabinet archive has just one file called core.sdb and has the same binary format as stage 1 (Windows executable with missing headers).

Figure 6. Contents of the CAB archive

Stage 2: core.sdb

The second-stage execution starts with the first environment checklist of blacklisted processes. The malware exits if it finds any of the following processes running (Figure 7).

```
"devenv.exe"
"wireshark.exe"
"vmacthlp.exe"
"procmon.exe"
"ollydbg.exe"
"idag.exe"
; "ImmunityDebugger.exe"
"windbg.exe"
"EHSniffer.exe"
"iris.exe"
"procexp.exe"
"filemon.exe"

sltp://setup.gohub.online:1108/setup.bin?id=128&sz=6c767d9e5592a3bd02b46fc4b5e89cfe&os=65542&ar=0
```

Figure 7. Blacklisted processes tested in stage 2 (top) and download URL format (bottom)

Stage 2 involves collecting the system's basic information: unique identifier, OS version number, and architecture. This information is then passed to the command-and-control (C&C) server to download the next-stage payload. An example of a download URL is in Figure 7 (bottom). The initial protocol identifier “sltp” is there most likely to confuse security researchers (similar to the use of wasm and the .sdb extension in stage 1). Communication is done via Windows Socket application programming interface (API), such as WSASocketA, BindIoCompletionCallback, and WSAIoctl among others.

The URL above is checked for sltp prefix. The domain and port are then extracted and used in socket communication. The communication is encrypted with RC4 cipher. The BindIoCompletionCallback API function defines the callback function, which takes care of the entire communication including encryption and decryption. The downloaded payload is loaded into memory and executed.

```
if ( BindIoCompletionCallback(v11, (LPOVERLAPPED_COMPLETION_ROUTINE)IOCompletionRoutine, 0) )
{
  a2 = 0;
  if ( wsa_ioctl(v6[13], 0xC8000006, &v34, 16, &a2, 4, &a3, 0, 0) != -1 )
```

Figure 8. BindIoCompletionCallbak with the IOCompletionRoutine

Stage 2: Custom filesystem and coredll loader

Stage 3 is basically a position-independent shellcode that searches inside its body for the marker “!rbx”, which is the beginning of RC4-encrypted data. The first 0x20 bytes of the binary block are used as a password, while the remaining bytes are the encrypted data. The decrypted blob also needs to be unpacked.

```
cmp     dword ptr [esi], 'xbr!'
jnz     loc_20E
```

Figure 9. !rbx marker within the stage 3 code

After unpacking, we obtained a file with an interesting structure: a custom ROM file system (romfs). In stage 3, “bin/i386/preload” and “bin/i386/coredll.bin” are extracted, merged, then executed in the next stage. Preload is basically a loader of coredll.bin. It maps the sections of loaded binary into memory, resolves APIs, rebases image, and jumps to the entry point.

rom1fs (custom romfs)

The previous stage is done to decrypt and unpack the binary blob, which leads to a file with the structure similar to a ROM file system disc. Searching for the „-rom1fs-„ signature led us to the description of romfs, which is a simple ROM file system format. The online documentation of romfs describes the layout of the file system in Figure 11 and 12.

```
2D 72 6F 6D 31 66 73 2D|B0 7C 29 00 00 00 00 00  | -rom1fs-°|).....
52 4B 49 00 00 00 00 00|00 00 00 00 00 00 00 00  | RKI.............
A0 05 00 00 5A 05 00 00|65 74 63 2F 6D 61 6C 77  |  ...Z...etc/malw
61 72 65 2E 6A 73 00 00|7B 0D 0A 20 20 20 20 22  | are.js..{..    "
```

Figure 10. The custom romfs

```
The layout of the filesystem is the following:

offset      content

            +---+---+---+---+
  0         | - | r | o | m |  \
            +---+---+---+---+      The ASCII representation of those bytes
  4         | 1 | f | s | - |  /   (i.e. "-rom1fs-")
            +---+---+---+---+
  8         |   full size   |      The number of accessible bytes in this fs.
            +---+---+---+---+
 12         |    checksum   |      The checksum of the FIRST 512 BYTES.
            +---+---+---+---+
 16         | volume name   |      The zero terminated name of the volume,
            :               :      padded to 16 byte boundary.
            +---+---+---+---+
 xx         |     file      |
            :   headers     :
```

Figure 11. romfs documentation describing the file system header

```
offset      content

        +---+---+---+---+
  0     | next filehdr|X|       The offset of the next file header
        +---+---+---+---+         (zero if no more files)
  4     |   spec.info   |       Info for directories/hard links/devices
        +---+---+---+---+
  8     |     size      |       The size of this file in bytes
        +---+---+---+---+
 12     |   checksum    |       Covering the meta data, including the file
        +---+---+---+---+         name, and padding
 16     | file name     |       The zero terminated name of the file,
        :               :       padded to 16 byte boundary
        +---+---+---+---+
 xx     | file data     |
        :               :
```

Figure 12. romfs documentation describing the file header

However, the malware creators did not follow the exact implementation of romfs. For instance, they included a few changes that made it impossible to mount the romfs in Linux and extract files easily. The file system header has a fixed size of 0x20 bytes. The first 0x10 bytes are the same as the one in the custom romfs, while the last 0x10 bytes are reserved for volume name, which is named RKI most likely as a shortcut for rootkit. The file header, however, is much more modified. Spec info and checksum fields are omitted; the file name is padded with just two 0x00 bytes, and the 16-byte padding boundary is not followed. These changes led us to write our own, custom romfs parser and extractor in Python. Running our parser, we extracted 38 files in the romfs.

```
magic = struct.unpack('<8s', fsdata[0:8] )
discsize = struct.unpack('<I', fsdata[8:12] )
checksum = struct.unpack('<I', fsdata[12:16] )
volumename = struct.unpack('<16s', fsdata[16:32] )

print magic
print discsize
print checksum
print volumename

offset = 0x20

while offset < discsize[0]:
    nextptr = struct.unpack('<I', fsdata[offset:offset+4] )
    size = struct.unpack('<I', fsdata[offset+4:offset+8] )[0]
    filename = struct.unpack('<32s', fsdata[offset+8:offset+40] )

    print("%08x"%nextptr)
    print("%08x"%size)
    filename_size = len(filename[0].split('\0')[0])
    filenamex = filename[0][0:filename_size]
    print( filenamex )

    ff = open( (filenamex.replace('/','_').replace('.','_').replace('\0','')), 'wb')
    ff.write( fsdata[offset + 8 + filename_size + 2: offset + 8 + filename_size + 2 + size])
    ff.close()

    offset = nextptr[0]

    if offset == 0:
        break
```

Figure 13. Custom romfs parser we used to extract files

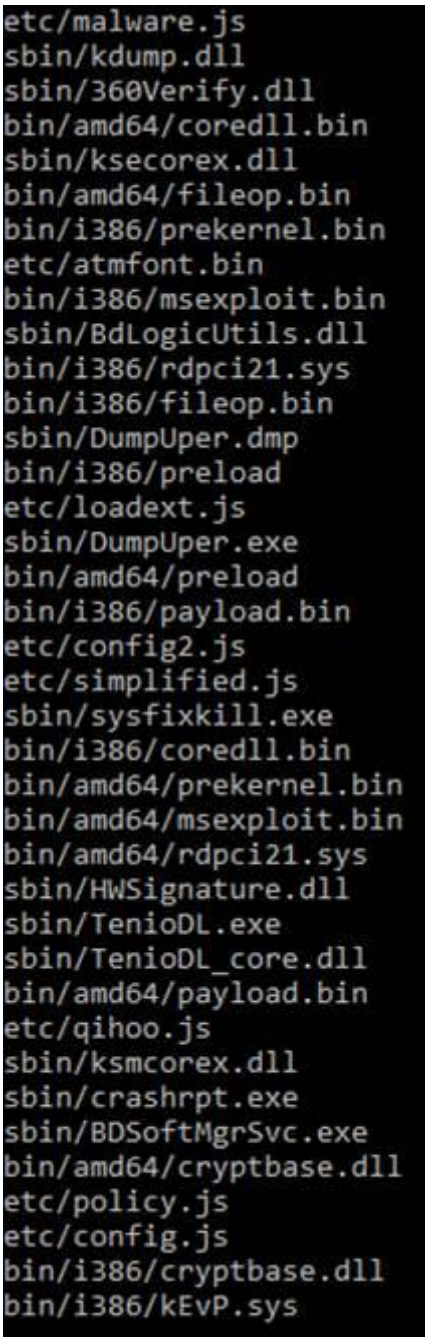

Figure 14. The romfs' contents

Stage 4: coredll.dll

coredll's main function is parsing a few configuration files from romfs and migrating itself into a different process based on the configuration (e.g., create a new process, inject itself into new process, execute itself in new process, terminate old process). The configuration files in romfs have the extension JS (not JavaScript in this case); the internal format of the configuration files is JavaScript Object Notation (JSON).

The first parsed configuration file is a simplified.js file. It has a section called "options" with value vmx_ignore, which is probably for debugging purposes. As shown in Figure 15, vmx_value=true means that the malware will ignore the presence of virtual machines and run even in a virtual environment. The malware.js file contains definitions (process names, company names, signatures) of several anti-malware products including KingSoft, Qihoo, Tencent, Baidu, Rising, HuoRong, MalwareDefender, and SSM.

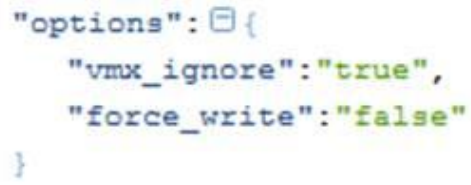

Figure 15. Options section in simplified.js file

Figure 16. Snapshot of antivirus (AV) products listed in malware.js's configurations

Based on the configuration specified in malware.js and processes found running on the target machine, coredll looks into another configuration file (policy.js) to determine what it does next. There are several

sections, one for each AV product listed in malware.js and a {default} section for machines with no AV installed.

Each section contains a list of commands that it can perform, e.g., what directory name should be created, which files from romfs should be extracted, and what parameters should be passed to the newly created processes during execution.

Note that the executables shown in Figure 17 are legitimate signed EXE files, usually signed by the respective AV vendor. These executables then load a DLL via the side-loading method. These DLLs will read the coredll contents via shared section object. With this step, the coredll execution flow is transferred to another process, which is usually signed by the manufacturer or currently running AV program. In the configuration file, the {%userdata} marker is replaced by the name of the shared section object, usually named STMxxxx. This section object can be shown with help of WinObj utility from SysInternals.


```
{
   "(default)":{
      "mk-dir":"SogouWP",
      "file-list":[
         "/sbin/crashrpt.exe",
         "/sbin/HWSignature.dll"
      ],
      "argv":"0 1 2 3 4 5 {%userdata}"
   },
   "Tencent":{
      "mk-dir":"Tencent",
      "file-list":[
         "/sbin/TenioDL.exe",
         "/sbin/TenioDL_core.dll"
      ],
      "argv":"{%userdata}"
   },
   "KingSoft":{
      "mk-dir":"KSoft",
      "file-list":[
         "/sbin/sysfixkill.exe",
         "/sbin/ksmcorex.dll",
         "/sbin/ksecorex.dll",
         "/sbin/kdump.dll"
      ],
      "argv":"{%userdata}",
      "script-file":"/etc/loadext.js"
   },
```


Figure 17. Code snippet showing the parsed policy.js

Figure 18. Shared section object named STMxxx as shown by the WinObj utility

The way the legitimate EXE file is signed also matters. Based on user privileges and other system settings, the execution may take one of several possible paths, and in some cases even uses embedded exploit files:

- Use OLE Automation (i.e., IShellWindows, IShellDispatch)
- Use ShellExecute API
- Create process using WMI
- Win7Elevate using fileop.bin and cryptbase.dll from ROMFS
- Payload.bin, prekernel.bin, msexploit.bin, atmfont.bin from ROMFS

After transferring coredll to another process, it executes the same binary but takes a different code path. When we parsed the config.js or config2.js configuration files, we found a few URLs:

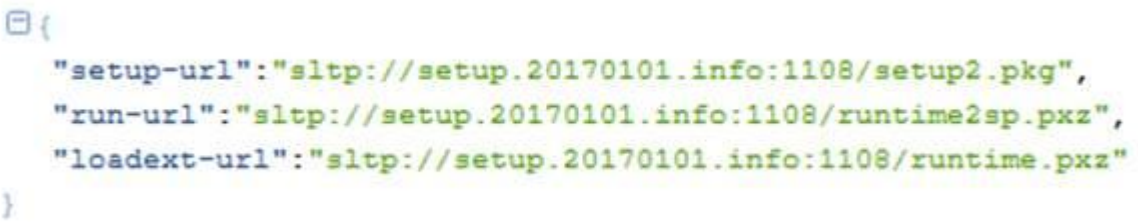

Figure 19. Parsed config.js configuration file showing URLs

In a similar vein, Windows sockets and RC4 are used during the C&C communication. The downloaded payload (setup2.pkg) is another romfs image; the setup.bin file in the binary gets executed first.

Stage 6: setup-url

The setup2.pkg file, once decrypted and unpacked, contains 11 files. The execution starts with the setup.bin file, which parses the config.js configuration file for update URLs (Figure 20, center). The setup2.pkg file is responsible for installing the bootkit into the infected machine. It is basically a continuation of the coredll setup after successfully passing some environment checks. The arksig.js file contains anti-rootkit program signatures that are used to hinder anti-rootkit tools.

```
setup.img
bin/amd64/kernel.sig
bin/i386/kernel.bin
bin/i386/kernel.sig
bin/amd64/kernel.bin
arksig.js
bin/amd64/dump.bin
bin/amd64/setup.bin
bin/i386/dump.bin
config.js
bin/i386/setup.bin
```

```
{
   "url": [
      "sstp://*.gatedailymirror.info/upd.pkg",
      "sstp://*.redteamshop.info/upd.pkg"
   ]
}
```

```
{
   "sig": [
      "gmer.pdb",
      "Win64AST.pdb",
      "View++_Driver64.pdb",
      "kEvP64.pdb",
      "TheArk.pdb",
      "kEvP.pdb",
      "View++_Driver.pdb",
      "SpyHunterDrv.pdb",
      "KmNife.pdb"
   ]
}
```

Figure 20. Contents of setup2.pkg romfs (top), the URLs found in the config.js configuration file (center), and the anti-rootkit file signatures in arksig.js (bottom)

Stage 6: run-url

After downloading a stream of data from a location specified in the run-url parameter, RC4 decryption and unpacking follows. We were able to obtain another romfs file system containing four files.

Code execution continues with loading and executing the subsystem binary, whose main task is to load and decrypt pgfs.pkg, which is another file that has a structure of a volume. This structure, however, is different from the romfs we encountered during our analysis. In this case, it downloads a mixed romfs file system.

This file system contains the following files: config.bsc, ccmain.cfg, ccmain.bin, and cloudcompute.api. As reported by other researchers, the binaries from this filesystem download and execute the Hidden Mellifera cryptocurrency-mining malware.


```
4D 69 78 65 64 20 52 4F|4D 46 53 00 00 00 00 00  | Mixed ROMFS.....
00 00 00 00 00 00 00 00|42 00 00 00 0A 00 00 00  | ........B.......
00 41 00 00 20 00 00 00|C0 03 00 00 00 41 00 00  | .A.. ...À....A..
30 00 00 00 C1 05 00 00|65 74 63 00 00 41 00 00  | 0...Á...etc..A..
24 00 00 00 C1 08 00 00|62 69 6E 00 00 81 00 00  | $...Á...bin.....
42 00 00 00 43 12 00 00|63 6F 6E 66 69 67 2E 62  | B...C...config.b
73 63 00 00 00 81 00 00|32 00 00 00 C3 16 00 00  | sc......2...Ã...
63 63 6D 61 69 6E 2E 63|66 67 00 00 00 41 00 00  | ccmain.cfg...A..
34 00 00 00 42 0B 00 00|61 6D 64 36 34 00 00 00  | 4...B...amd64...
00 41 00 00 34 00 00 00|C2 0E 00 00 69 33 38 36  | .A..4...Ã...i386
00 00 00 00 00 81 00 00|80 1B 00 00 45 1A 00 00  | ........€...E...
63 6C 6F 75 64 63 6F 6D|70 75 74 65 2E 61 70 69  | cloudcompute.api
00 00 00 00 00 81 00 00|00 FC 01 00 C3 D2 01 00  | .........ü..ÃÒ..
63 63 6D 61 69 6E 2E 62|69 6E 00 00 00 81 00 00  | ccmain.bin......
00 0F 00 00 C5 9A 21 00|63 6C 6F 75 64 63 6F 6D  | ....Å■!.cloudcom
70 75 74 65 2E 61 70 69|00 00 00 00 00 81 00 00  | pute.api........
80 C0 01 00 03 8B 22 00|63 63 6D 61 69 6E 2E 62  | €À...■".ccmain.b
69 6E 00 00 6A 01 00 00|82 A3 78 38 36 91 BA 2F  | in..j...■£x86'º/
```

Figure 21. List of files in the runtime romfs (top), and a hexadecimal view of the mixed romfs file (bottom)

```
{"id":1,"jsonrpc":"2.0","method":"login","params":{"login":"b0644fe5-8fb3-2a77-c6ea-3956a630459d","pass":"x","agent":"MinGate/5.1"}}
{"id":1,"jsonrpc":"2.0","error":null,"result":{"id":"70bdd273-d886-4985-8869-da59dcc57a02","job":
{"blob":"0707b8e5d7da050c7354e4ec83ce3b36e2d9718e2d95397f99df9bcd735669495125b4528a9b370000000074dcd3fbf446737c038d3256e4b8a96bb4343a0
e5cf130b68216ad5ddbd2e6be02","job_id":"DdE8fBM0C4WkCnUwld8a/wmNjzJo","target":"37894100","id":"70bdd273-d886-4985-8869-
da59dcc57a02","algo":"cryptonight/1"},"status":"OK"}}
{"method":"submit","id":2,"params":{"nonce":"7f070000","id":"70bdd273-d886-4985-8869-da59dcc57a02","job_id":"DdE8fBM0C4WkCnUwld8a/
wmNjzJo","result":"2fb204117585ef550aea88e676d302ca49205ed003aae75a51e82170b8c42f00"},"jsonrpc":"2.0"}
{"id":2,"jsonrpc":"2.0","error":null,"result":{"status":"OK"}}
{"method":"submit","id":3,"params":{"nonce":"cd070000","id":"70bdd273-d886-4985-8869-da59dcc57a02","job_id":"DdE8fBM0C4WkCnUwld8a/
wmNjzJo","result":"6d3f26b76e1e9ac84ad138bbf0299ebea1b3ad65adf6b0b988304abd03dc3100"},"jsonrpc":"2.0"}
{"id":3,"jsonrpc":"2.0","error":null,"result":{"status":"OK"}}
```

Figure 22. Code snippet showing Hidden Mellifera communicating with the mining pool server

Indicators of Compromise (IoCs)

Malicious URLs and IP Address related to Underminer:

- pop[.]tz365[.]vip (malicious advertising server)
- hxxp://144.202.87[.]106/index.php (malvertising URL)
- hxxp://103.35.72[.]231/ip.php (exploit kit redirect URL)
- 103[.]35[.]72[.]223 (Underminer exploit kit server)

Malicious domains related to Hidden Mellifera:

- setup[.]20170101[.]info
- setup[.]gohub[.]online
- gatedailymirror[.]info
- redteamshop[.]info
- ask[.]thesupporthelp[.]com
- stratum+tcp://data[.]supportithelp[.]com:8080

Related Hashes (SHA-256):

File	Hash	Trend Micro Detection
Stage 1 wasm	ae2f07d390eda32458e07f1e8310d8539f88bd6b2476a9c3f170667005dcc563	TROJ_DLOADR.AUSUMP
Stage 2 wasm	ccd77ac6fe0c49b4f71552274764ccddcba9994df33cc1240174bcab11b52313	
core.sdb	c1a6df241239359731c671203925a8265cf82a0c8c20c94d57a6a1ed09dec289	TROJ_DLOADR.AUSUMQ
/bin/amd64/coredll.bin	11e23045ea347bb3b6d2e7f2da826e399a56a18804d97fb30d137b9b21ff1478	TROJ_MALOAD.QFKG
/bin/amd64/cryptbase.dll	a2db6542b957a7d03e28fb09193c2e1f2bb8a8305ca435ba0fa972aa6240ede2	
/bin/amd64/fileop.bin	d31abe88e1e6b8b84ffd99c8962cc194c9f53eb633647ac6d23c35699d4d6b98	
/bin/amd64/msexploit.bin	a8dcb26601e2a51520a83105304edc42d5c66647b09622a8acfb285c2f690ab6	
/bin/amd64/payload.bin	33f0691d79fa69c3d836c2ce68db4fbdf73bdb942b6cc61fb18b898f7458bf1b	
/bin/amd64/prekernel.bin	07b26d00bdac8fc07ef5662d536645532c30e296c8e8f056602089595a00e24b	
/bin/amd64/preload	da16c6e5f2425aa47cac79b15a33e55ee05d93d5dddb3dcde42bb57fd9de9ce7	TROJ_MALOAD.QFKG
/bin/amd64/rdpci21.sys	72134f4da6941b87e9c06d045653d2a012da7a2bab299dcc641518e8e8b56f96	

File	Hash	Trend Micro Detection
/bin/i386/coredll.bin	a004b85f939e61447e40973581ca7b2ed9aea5b154f81c4beb97f84661ba73eb	TROJ_MALOAD.QFKG
/bin/i386/cryptbase.dll	53dd5d76190cf32037ca51105cb220276b24d6d695b61d00926372ca16a5de2b	
/bin/i386/fileop.bin	a1671fe30862e056148b1d0d1302fe7c9925eaef805fd665dc498ce052a448f9	
/bin/i386/kEvP.sys	a795deaa2d1c1f2d9426a8c28791111e0192ffad14d086b51bc61c8e16008b63	
/bin/i386/msexploit.bin	e19f8d3b2accc981c20ace2ed1a305828b4842920286b6e376a71f65b3e8ada0	
/bin/i386/payload.bin	62d31ce9aad0d1ad22f3836f3098843674c1ced075554cbde06cda74fbb71e09	
/bin/i386/prekernel.bin	a5da0061dbde4bbfb31c3de3036c3028f50d21dffd9e3aa0e2af201e56b3d239	
/bin/i386/preload	f8ff6f15fe09ad16b234db9e3a746e468abe8b5a13a047b797cffeb404b872c7	
/etc/atmfont.bin	3d9c17ebf675edbe63642387010b2dfc39eb22b6526da86b59a5b4a5d78f1b4d	TROJ_EXPLOYT.TIDAICR
/etc/config2.js	c1259a123dabf20bb4ea580fee986850c0ad03a2fb917829142cdb2404cf645a	TROJ_MALOAD.CFG
/etc/config.js	864244757126d79f50ee1a3e20164d441bf1462b1b92bf564d3486cdf0f2194d	
/etc/policy.js	a62af9e220c47f37fdc7c4c5527c5fc744b82d9f04e3e5ee523efa98be89fdee	
/sbin/360Verify.dll	f18e8107fdcc7173f5e0956099e6d617da9582b2991fec3b9e94adf77638df58	TROJ_MALOAD.QFKF
/sbin/BdLogicUtils.dll	fb6ffad15ea2f0a38614b6140c0a66f9d32f1f42f4beb96a574f44a4b5d91f51	
/sbin/HWSignature.dll	68c08445a79632309b52ba2d51f1a2cdf7c4094b27ebe220506038056983c961	
/sbin/kdump.dll	3af1bbc3517155160aa1c9ed25035e0f589293feb363d431caa615d400c6708d	
/sbin/TenioDL/core.dll	0cf328c9c21453fbb588496e74387901501e90c44a1ff98c0ca6f328da03991f	
/bin/amd64/dump.bin	a9bacb33b6e29e3a69f67d2c8cf3b8f5ab84c8a395668b553795fb6bd0b4b75b	TROJ_MALOAD.QFKG
/bin/amd64/kernel.bin	7f12359d4d7aa937208e9812c977b4b396a096bd7675c57dbb8dd1b2a468bd56	
/bin/amd64/kernel.sig	717c9c82fd5c4b7fb09310a57710d333843d82b81f2baf03c38076a555f2fe34	TROJ_MALOAD.CFG
/bin/amd64/setup.bin	1766ae5d57d60924c71832a1f3ce3fdc15e0410ed6bdd3e1a198cb090d001ce2	TROJ_MALOAD.QFKG
/bin/i386/dump.bin	0f2d76b68be299aabaf7041d482d0a2a0bc5d7fb9b90282582334eb67f439875	
/bin/i386/kernel.bin	395c9c64426721527effa384afcde57f10e199e6b29c3b3bb93574609a4a8b01	
/bin/i386/kernel/sig	64e7cf971e298988495d88a8014ce8d2728b05ec0b1bf77cb2b731f51e726ae7	TROJ_MALOAD.CFG
/bin/i386/setup.bin	83df8971de715d318b6667ec0a94d66b5bc87f6d6864047647ae87c55fcaed6d	TROJ_MALOAD.QFKG

File	Hash	Trend Micro Detection
/config.js	4c0c64e3ab19945d10da51338c01db10b22b6f899037192da4e4059cc5c1b2ac	TROJ_MALOAD.CFG
/setup.img	79548d4a17cf0a38ac66ca216961cf92f4a3648821d37700cd8d09d39a91b729	
/bin/amd64/subsystem	6b8d868b373748fca5c3a7c76b50b686126d430ce6ff35c0ef9907800b81b805	TROJ_MALOAD.QFKG
/bin/i386/subsystem	56e1fafe7b81aab17765c5bd080f93cf2366553ac9dcef6dc75f895ec5a859ea	
/pgfs/pgfs.pkg	03662ac576ec50d388f87055d2f4295f56f7e682e13703d452d14c2a7f9cb196	COINMINER_MMXMR.C-ENC

Securing Your Journey to the Cloud

Trend Micro Incorporated, a global leader in security software, strives to make the world safe for exchanging digital information. Our innovative solutions for consumers, businesses and governments provide layered content security to protect information on mobile devices, endpoints, gateways, servers and the cloud. All of our solutions are powered by cloud[-]based global threat intelligence, the Trend Micro™ Smart Protection Network™, and are supported by over 1,200 threat experts around the globe. For more information, visit www.trendmicro.com.

Created by:

TrendLabs

Global Technical Support & R&D Center of TREND MICRO