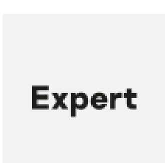
Gootkit: the cautious Trojan

securelist.com/gootkit-the-cautious-trojan/102731/

Authors



Anton Kuzmenko

Gootkit is complex multi-stage banking malware that was discovered for the first time by Doctor Web in 2014. Initially it was distributed via spam and exploits kits such as Spelevo and RIG. In conjunction with spam campaigns, the adversaries later switched to compromised websites where the visitors are tricked into downloading the malware.

Gootkit is capable of stealing data from the browser, performing man-in-the-browser attacks, keylogging, taking screenshots and lots of other malicious actions. Its loader performs various virtual machine and sandbox checks and uses sophisticated persistence algorithms. In 2019, Gootkit stopped operating after it experienced a data leak, but has been active again since November 2020.

Gootkit's victims are mainly located in EU countries such as Germany and Italy. In this article we analyze a recent sample of Gootkit.

Technical Details

Gootkit consists of a (down)loader component written in C++ and the main body written in JS and interpreted by Node.js. The main body is a modular framework, containing registration, spyware, VMX detection and other modules.

Loader

The sample (MD5 97713132e4ea03422d3915bab1c42074) is packed by a custom-made multi-stage packer which decrypts the final payload step by step. The last stage is a shellcode that decrypts the original loader executable and maps it into memory. After mapping, the original entry point is called. Hence, we can easily unpack the original executable and analyze it. We detect the Gootkit loader with the verdicts listed in the table below.

MD5	SHA-1	Verdict
97713132e4ea03422d3915bab1c42074	a90c6e7c5650e73ceb0b329fa8c78045632100ee	Trojan-Downloader.Win32.Injecter
27626f2c3667fab9e103f32e2af11e84	6e9e30c699c7111089fe364ce47f1dc05c8bc703	HEUR:Trojan.Win32.Generic

Most of the strings are encrypted using XOR encryption and are decrypted at runtime. No other techniques are used to complicate static analysis.

```
v94 = 593372721;
v95 = (LPCWSTR)85330738;
v96 = 825630810;
v97 = 692127773;
LOWORD(v98) = 12837;
BYTE2(v98) = 114;
*(_DWORD *)v93 = 1311983986;
*(_WORD *)&v93[4] = 25939;
v22 = GetProcessHeap();
v23 = (const CHAR *)HeapAlloc(v22, 8u, 0x14u);
v24 = 0;
lpProcName = v23;
v99 = 0;
*(_DWORD *)v23 = 0;
*((_DWORD *)v23 + 1) = 0;
*((_DWORD *)v23 + 2) = 0;
*((_DWORD *)v23 + 3) = 0;
*((_DWORD *)v23 + 4) = 0;
P = (PVOID)((char *)&v94 - v23);
do
{
  v25 = (CHAR *)&v23[v24];
  v26 = (v23[v24 + (_DWORD)P] ^ v93[v24 % 6]) - GetLastError();
  v27 = GetLastError();
  v23 = lpProcName;
  v28 = v26 + v27;
  v24 = v99 + 1;
  *v25 = v28;
  v99 = v24;
}                                             // CommandLineToArgvW
while ( v24 < 19 );
```

String decryption

However, to make dynamic analysis more difficult, the Gootkit loader employs lots of different methods to detect virtual environments or debuggers. If any of the virtual machine checks succeed, the loader enters an infinite loop.

```
v0 = PathFindFileNameW(module_name);
v1 = v0;
v2 = wstr_crc(v0, -1);
v3 = 0xBC136B46;                        // SAMPLE.EXE
v9[0] = 0xD84A20AC;                     // SANDBOX.EXE
v4 = 0;
v5 = v2;
v9[1] = 0xEED889C4;                     // MALWARE.EXE
v9[2] = 0x58636143;                     // TEST.EXE
v6 = (int)(v1 + 1);
v9[3] = 0xC0F26006;                     // BOT.EXE
v9[4] = 0x8606BEDD;                     // KLAVME.EXE
v9[5] = 0xE8CBAB78;                     // MYAPP.EXE
v9[6] = 0x2AB6E04A;                     // TESTAPP.EXE
v9[7] = 0x31E6D1EA;                     // ?
v10 = 0;
do
{
  v7 = *v1;
  ++v1;
}
while ( v7 );
if ( (unsigned int)(((int)v1 - v6) >> 1) < 0x20 )// length is less than 0x20
```

Sample name check

Full list of VM detection techniques used by the malware:

Check	**Prohibited value**
CRC32 of sample name	0xBC136B46, 0xD84A20AC, 0xEED889C4, 0x58636143, 0xC0F26006, 0x8606BEDD, 0xE8CBAB78, 0x2AB6E04A, 0x31E6D1EA
GetModuleHandle	dbghelp.dll, sbiedll.dll
GetUserName	CurrentUser, Sandbox
GetComputerName	SANDBOX, 7SILVIA
HKEY_LOCAL_MACHINE\HARDWARE\DESCRIPTION\SystemBiosVersion	FTNT1, INTEL-604000, SMCI, QEMU, VBOX, BOCHS, AMI, SONI
HKEY_LOCAL_MACHINE\HARDWARE\DESCRIPTION\VideoBiosVersion	VirtualBox

HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\SystemBiosVersion	55274-640-2673064-23950 (Joe Sandbox), 76487-644-3177037-23510 (CWSandbox), 76487-337-8429955-22614 (Anubis Sandbox)
HKEY_LOCAL_MACHINE\HARDWARE\DESCRIPTION\System\CentralProcess\0\ProcessorNameString	Xeon
_MEMORYSTATUSEX. ullTotalPhys	Less than 2100000000
UuidCreateSequential (this function is based on computer MAC address so return value is used to determine whether trojan is running in sandbox or not)	0xF01FAF00 (Dell Inc.), 0x505600 (VMWare, Inc.), 0x8002700 (PCS System Technology GmbH), 0xC2900 VMWare, Inc.), 0x56900 (VMWare, Inc.), 0x3FF00 (Microsoft), 0x1C4200 (Parallels), 0x163E00 (XenSource)
CRC32 of running process names	0xAEA3ED09, 0x2993125A, 0x3D75A3FF, 0x662D9D39, 0x922DF04, 0xC84F40F0, 0xDCFC6E80

Execution flow

When the sample starts, it checks the command line arguments. The available arguments are listed below:

Argument	Description
–client	no handler
–server	no handler
–reinstall	iterate over running processes (where process is a loop variable) and kill all processes where *process.pid* is not equal to current process PID and *process.name* equals current filename. After that, copy self and run via CreateProcessW
–service	set environment variable USERNAME_REQUIRED=TRUE
–test	stop execution
–vwxyz	download main body from C&C

After the command line arguments are handled, the sample checks if it’s running inside a virtual machine or being debugged. If not, it decrypts the configuration and starts four threads.

```
THREAD_HANDLES_LIST = CreateThread(0, 0, update_from_c2, module_name, 0, 0);
*(&THREAD_HANDLES_LIST + 1) = CreateThread(0, 0, browser_inj, v86, 0, 0);
*(&THREAD_HANDLES_LIST + 2) = CreateThread(0, 0, persistence_service, v86, 0, 0);
v87 = CreateThread(0, 0, stop_switch, Parameter, 0, 0);
CloseHandle(v87);
```

Thread start routine

- **Update_from_c2**
  The first thread that is started tries to download a loader update from <CnC host>/rpersist4/<crc>, where <CnC host> is a command-and-control server address and <crc> is the CRC32 of the first 0x200 bytes of the current file in decimal format.
- **Browser_inj**
  The thread decrypts two embedded MZPE executables (x64 and x86 DLLs), iterates over the running processes and tries to inject the decrypted DLLs into the process memory of the designated process using the NtCreateSection/NtMapViewOfSection API. Matching of the process name is done by calculating the CRC32 value of the process name. For a list of supported browsers, see the table below.

CRC32	Browser name
0xC84F40F0	Chrome
0x662D9D39	Firefox
0x922DF04	Internet Explorer
0x2993125A	Microsoft Edge (MicrosoftEdgeCP.exe)
0x3D75A3FF	Opera
0xDCFC6E80	Safari
0xEB71057E	unknown

The injected code is called from the main body web injection and traffic sniffing routines to perform a man-in-the-browser attack. To do so, the code patches standard browser functions responsible for certificate validation to allow self-signed certificates. As a result, attackers are able to inject custom JS code and modify or redirect traffic.

- **Persistence_service**
  If a sample is running under LOCAL_SYSTEM account, the Gootkit persistence mechanism abuses the pending GPO Windows feature. When a user modifies Pending GPO registry values, he/she has to specify the following parameters:
  - count – count of pending GPOs;
  - path1, path2, … – path to the special .inf file that contains instructions on how to load GPO;
  - Section1, Section2, … – name of the section from the INF file.

  So Gootkit creates an .inf file in the same directory as the sample and writes the following values to the Software\Microsoft\IEAK\GroupPolicy\PendingGPOs registry key:

  - count – 0x1
  - path1 – .inf file location
  - Section1 – DefaultInstall

```
// [Version]
// signature = "$CHICAGO$"
// AdvancedINF = 2.5, "You need a new version of advpack.dll"
//
// [DefaultInstall]
// RunPreSetupCommands = <random string>:2
//
// [<random string>]
// <sample path>
```

  INF file content

  Now explorer.exe will load the Group Policy Objects (GPO) whenever it is loaded. Gootkit creates a pending GPO for the Internet Explorer Administration Kit (IEAK), which points directly at the INF file. When explorer.exe is loaded at runtime, it will execute the [DefaultInstall] inside the created file, which will run the Gootkit executable.

  If the sample is running under another account, it creates a service with a random name chosen from %SystemRoot%, copies itself into the %SystemRoot% folder with the chosen name and deletes itself from the disk.

- **Stop_switch**
  The thread looks for a file named uqjckeguhl.tmp in the \AppData\Local\Temp and \Local Settings\Temp folders. When the file is found, the malware will stop.

Main body download

Before downloading the main body from the C&C, the loader tries to find registry keys with the following format: *HKCU\Software\AppDataLow\<pr_string>_<i>*, where i is a number starting from 0 and pr_string is a pseudo-random string generated when the bot starts. Generation is based on the victim's PC parameters, so the same value is generated for the same PC each time.

Each key contains a maximum chunk of 512,000 bytes (500KB) of encrypted data. If the aforementioned keys were found, their contents will be saved in a newly allocated buffer (used for decryption and decompression). The buffer is then decrypted using the same function used for decrypting the configuration, after which the buffer is decompressed.

After the unpacking routine, the loader will download the main body from the C&C, calculate its CRC32 and compare it with the registry payload CRC (if one exists). If the CRCs are different, the loader will execute the newer version downloaded from the C&C. The C&C server will not send the DLL module without the appropriate UserAgent header that is hardcoded into the sample. The current hardcoded value is: Mozilla/5.0 (Windows NT 6.1; Win64; x64; rv:25.0) Gecko/21006101 Firefox/25.0.

```
result = 0;
ctr = 0;
result = 0;
i = 0;
key = 34;
ctr = 0;
if ( size > 0 )
{
  do
  {
    if ( i >= 0x400 )
    {
      block_ptr = &data[result];
      result += i;
      qmemcpy(block_ptr, tmp_buf, i);
      i = 0;
    }
    tmp_buf[i++] = key ^ data[ctr];
    key += 3 * (ctr++ % 133);
  }
  while ( ctr < size );
  if ( i )
    qmemcpy(&data[result], tmp_buf, i);
}
return result;
```

Decrypt function

Main body

The main body (MD5 20279d99ee402186d1e3a16d6ab9398a, verdict HEUR:Trojan.Win32.Generic) is a Node.js interpreter with bundled encrypted JS files. On startup, the main body decrypts the JavaScript files using an RC4-like algorithm with hardcoded keystream.

Information about the embedded modules is stored in an array of special file structures that have the following format: BYTE* name_pointer, BYTE* encrypted_data, DWORD data_size, DWORD encr_flag. These structures are used within the decryption routine that reads **data_size** bytes starting from **encrypted_data**. This routine decrypts **encrypted_data** if **encr_flag** is set and writes the result into a file with name ***name_pointer**. The decryption routine iterates over all entries in the file information array. Then the decryption execution is transferred to the Node.js interpreter.

```
.data:104F64B0 C8 6F 47 10 28 25 4C+files         file_data <offset aNode, offset unk_104C2528, 118Eh, 1>; 0
.data:104F64B0 10 8E 11 00 00 01 00+                                 ; DATA XREF: sub_10054286+26↑r
.data:104F64B0 00 00 8C 84 47 10 30+                                 ; sub_10054286+32↑o ...
.data:104F64B0 A3 4E 10 C3 03 00 00+              file_data <offset aLinklist, offset unk_104EA330, 3C3h, 1>; 1 ; "sm
.data:104F64B0 01 00 00 00 98 84 47+              file_data <offset aAssert, offset unk_1051A810, 0E33h, 1>; 2
.data:104F64B0 10 10 A8 51 10 33 0E+              file_data <offset aConsole, offset unk_104B2920, 529h, 1>; 3
.data:104F64B0 00 00 01 00 00 00 A0+              file_data <offset aBuffer, offset unk_104D8118, 1533h, 1>; 4
.data:104F64B0 84 47 10 20 29 48 10+              file_data <offset aConstants, offset unk_104FF710, 285h, 1>; 5
.data:104F64B0 29 05 00 00 01 00 00+              file_data <offset aChildProcess, offset unk_104EE430, 28A4h, 1>; 6
.data:104F64B0 00 EC 60 47 10 18 81+              file_data <offset aCrypto, offset unk_104E8F88, 11E9h, 1>; 7
.data:104F64B0 4D 10 33 15 00 00 01+              file_data <offset aDgram, offset unk_10504E08, 0E5Ah, 1>; 8
.data:104F64B0 00 00 00 60 6F 47 10+              file_data <offset aCluster, offset unk_104E22F0, 18ADh, 1>; 9
.data:104F64B0 10 F7 4F 10 85 02 00+              file_data <offset aDns_0, offset unk_104F0DD8, 0CE0h, 1>; 0Ah
.data:104F64B0 00 01 00 00 00 A8 84+              file_data <offset aFreelist, offset unk_104D0E90, 393h, 1>; 0Bh
.data:104F64B0 47 10 30 E4 4E 10 A4+              file_data <offset aEvents_0, offset unk_104BB638, 0AE0h, 1>; 0Ch
.data:104F64B0 28 00 00 01 00 00 00+              file_data <offset aDomain, offset unk_104BFC50, 0D1Ah, 1>; 0Dh
.data:104F64B0 8C 6F 47 10 88 8F 4E+              file_data <offset aHttp_0, offset unk_104B8B78, 5EDh, 1>; 0Eh
.data:104F64B0 10 E9 11 00 00 01 00+              file_data <offset aHttpAgent, offset unk_104D2E10, 0BD0h, 1>; 0Fh
.data:104F64B0 00 00 B8 84 47 10 08+              file_data <offset aHttpCommon, offset unk_105128B8, 0B49h, 1>; 10h
.data:104F64B0 4E 50 10 5A 0E 00 00+              file_data <offset aHttpClient, offset unk_104CABD8, 1649h, 1>; 11h
```

File information array

The array contains 124 encrypted files, both Node.js system libraries and open-source packages, and malware modules. Strangely enough, the JS entry point is a file named malware.js.

Malware.js initializes global bot variables, collects saved cookies (IE, Firefox, Chromium) and iterates over a list of servers to find an available C&C.

When the malware finds a C&C server, it launches an infinite loop that listens to different internal malware events (some routines like cookie collection start without C&C request upon bot startup) and sends the collected data to the C&C via special formatted packets. The malware also listens to the C&C commands and invokes the appropriate handler on each command. To communicate with the modules, the malware uses following packet types:

Internal name	Description
SLAVE_PACKET_API_TAKESCREEN	Send screenshot to C&C
SLAVE_PACKET_MAIL	Send received email info
SLAVE_PACKET_LOGLINE	Send log
SLAVE_PACKET_LSAAUTH	Send authentication credentials
SLAVE_PACKET_PAGE_FRAGMENT	Send web injects data
SLAVE_PACKET_FORM	Send grabbed form data
SLAVE_PACKET_LOCAL_VARS	Send local bot variables
SLAVE_PACKET_SECDEVICELOG	Send secure device event log
SLAVE_PACKET_KEYLOG	Send keylogger data
SLAVE_PACKET_WINSPYLOG	Send current active window

There are six types of internal event handlers and corresponding packet formats.

```
const P_SOCKS = 0;
const P_PING = 1;
const P_FS = 2;
const P_REGISTRATION = 3;
const P_SPYWARE = 4;
const P_CMDTERM = 5;

var packetParsers = {};
var procolPacketBuilders = {};
var protocolDispatchers = {};

protocolDispatchers[P_SOCKS] = 'client_proto_socks';
protocolDispatchers[P_PING] = 'client_proto_ping';
protocolDispatchers[P_FS] = 'client_proto_fs';
protocolDispatchers[P_REGISTRATION] = 'client_proto_registration';
protocolDispatchers[P_SPYWARE] = 'client_proto_spyware';
protocolDispatchers[P_CMDTERM] = 'client_proto_cmdterm';
```

Event handlers

The general packet structure is as follows:

- Length + 8 (4 bytes)
- Packet magic (0xEDB88320 XOR length+8)
- Packet data (different for each package type, serialized using protobuf)
- Packet magic

```
var packetLength = new Buffer(4);
var packetMagic = new Buffer(4);
var protoMagic = 0xEDB88320;
var maxChunkSize = 4096;
packetLength.writeUInt32BE(packet.length + 8);
packetMagic.writeUInt32BE(ToUint32(protoMagic ^ packet.length));
self.push(packetLength);
self.push(packetMagic);
if (packet.length > maxChunkSize) {

    for (let i = 0; i < packet.length; i+=maxChunkSize)
    {
        self.push(
            packet.slice(i, Math.min(i + maxChunkSize, packet.length))
        );
    }
} else {
    self.push(packet);
}
self.push(packetMagic);
```

Packet generation routine

Kaspersky products detect this family as Trojan-Downloader.Win32.Injecter, HEUR:Trojan.Win32.Generic, Trojan-Downloader.Win32.Gootkit, Trojan-Banker.Win32.Gootkit. All the details, IoCs, MITRE ATT&CK Framework data, Yara rules and hashes related to this threat are available to the users of our Financial Threat Intelligence services. To learn more about threat hunting and malware analysis, check out expert training by Kaspersky's GReAT.

Indicators of compromise

Main body (same since 2019)
20279d99ee402186d1e3a16d6ab9398

Loader
5249c568fb2746786504b049bbd5d9c8
97713132e4ea03422d3915bab1c42074
174A0FED20987D1E2ED5DB9B1019E49B
27626f2c3667fab9e103f32e2af11e84

Domains and IPs
kvaladrigrosdrom[.]top
scellapreambulus[.]top
lbegardingstorque[.]com
kerymarynicegross[.]top
pillygreamstronh[.]com

- Financial malware
- Malware Descriptions
- Malware Technologies
- Trojan
- Trojan Banker

Authors

Expert Anton Kuzmenko

Gootkit: the cautious Trojan

Your email address will not be published. Required fields are marked *