Project

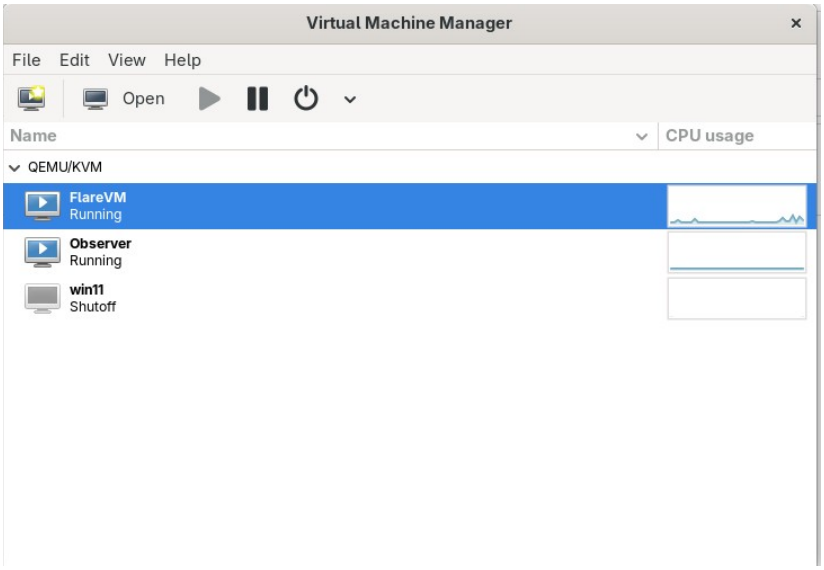
Class #

01/01/0001

First Last

Environment Setup

Overview



Host:

- OS: Arch Linux
- Architecture: amd64
- Hypervisor: QEMU/KVM
- GUI: virt-manager

Victim VM:

OS: Windows 11 Pro
Architecture: amd64
Disk: 86 GB VirtIO
CPU: 4 cores
RAM: 8 GB
IP: 172.29.1.195
Tools: FLARE-VM

Analysis VM:

OS: Arch Linux
Architecture: amd64
Disk: 64 GB VirtIO
CPU: 4 cores
RAM: 4 GB
IP: 172.29.1.205
Tools: Ghidra, INetSim

Network:

Mode: Isolated
Address: 172.29.1.1
Mask: 255.255.255.0

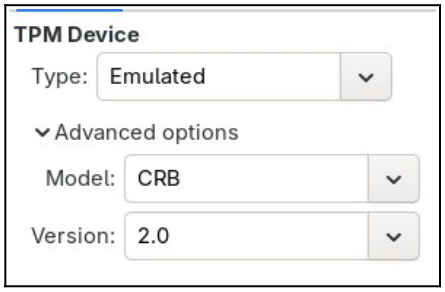
Malware:

Source: theZoo
Name: Zeus
Year: 2013

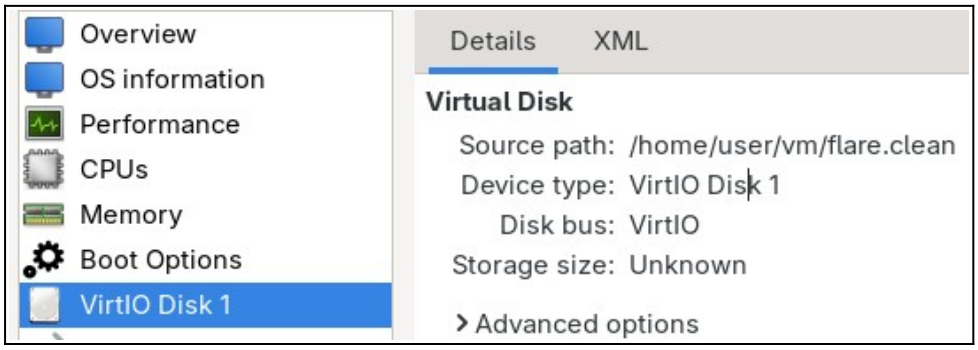
Victim VM

Since the malware I will be analyzing is made for Windows, the victim virtual machine will have to be running Microsoft Windows to make dynamic analysis possible. I chose the latest version of Windows 11 since that is what I had on hand.

When configuring the Windows 11 VM inside of virt-manger (a gui front end to QEMU/KVM), there where a few settings I had to modify. First, Windows 11 requires a Trusted Platform Module (TPM). This can be configures within virt-manager by adding a TMP module component. However, for it to function, I also needed to install the swtpm package on the host machine.

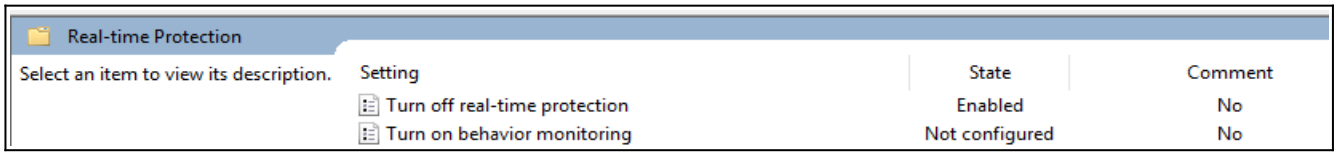


The only other major change was switching the disk bus type from the default SATA to VirtIO. It is a good change which improves performance in the VM. That being said, it requires additional drivers — which I had to install during the Windows installation; since the drive does not otherwise show up. Both the disk drives and guest vm drives which I installed can be found here: https://github.com/virtio-win/virtio-win-pkg-scripts.

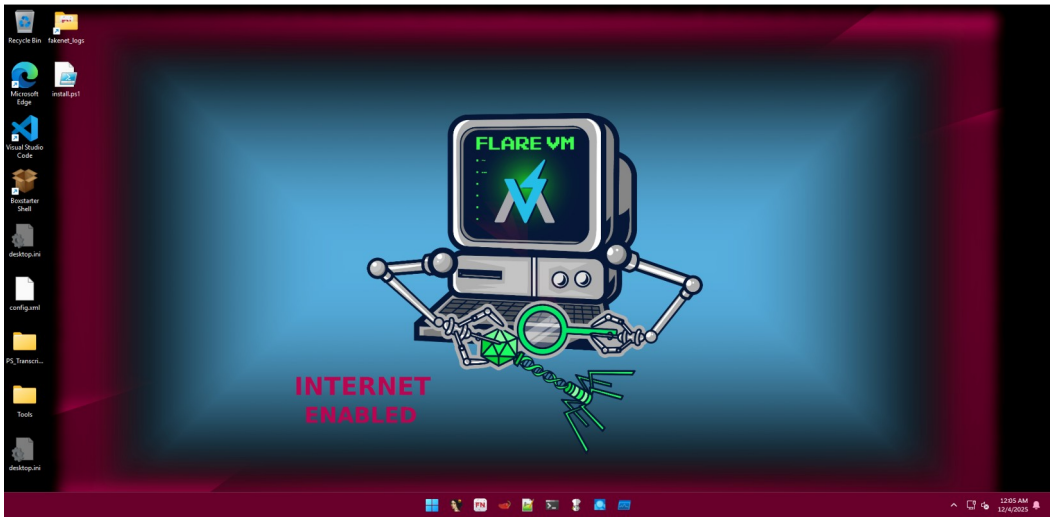


The victim VM needed certain tools to make dynamic analysis possible. Instead of manually installing them like I did for other class assignments, I decided to use FLARE-VM. It is a convenient script which setups a great environment for reverse engineering.

While the “install.ps1” script provided by FLARE-VM does most of the heavy lifting, there were still a few prerequisites I had to handle myself. Mainly turning off Windows Updates and Windows Defender. Both of these changes were achieved through Group Policy – which ensures they do not just turn back on with the next reboot. The FLARE-VM github provides resources which detail the exact steps required.



The most important step to not overlook is that “MsMpEng.exe” needs to be suspended before making changes to the group policy dealing with Windows Defender. Otherwise, things just revert back after a reboot.



After FLARE-VM installation completed, the desktop started looking as if the computer is already infected.

Overall, while FLARE-VM did most of the heavy lifting in terms of installing the necessary tools, I would likely not use it again If I had to remake the victim VM.

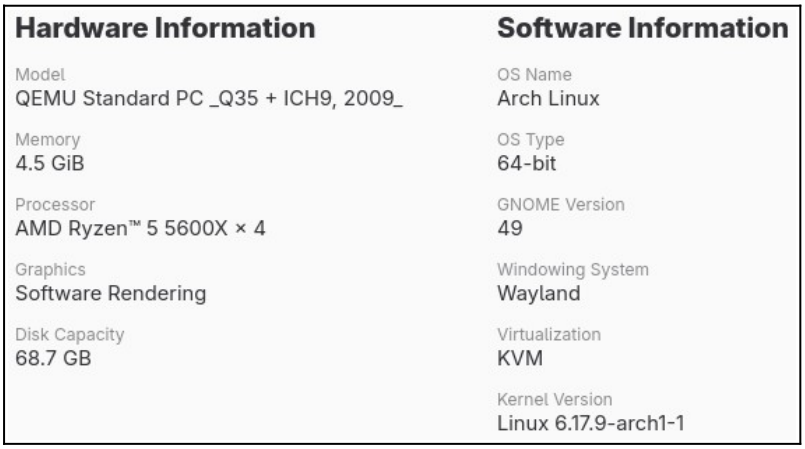
It largely comes down to speed — It took multiple hours to install on the slow dorm internet connection. It would have been quicker for me to manually install just the tools I will actually be using.

Analysis VM

The reason for a separate analysis VM, despite the fact that the victim VM already comes with all the static analysis tools, is that the victim VM is an untrusted and unstable environment. Since it will be used for dynamically analyzing live malware, I will be constantly rolling it back to clean snapshots. Since I want all the static analysis progress (like renamed variables) to remain, it makes sense to do all that work in a separate and more trusted VM.

For security reasons, it made sense to go with a Linux operating system. This would prevent the possibility of accidentally running the malware on the analysis VM.

I originally tried using REMnux (basically FLARE-VM for linux). However, it's use of outdated Ubuntu 20.04 proved to be a bigger hassle than the worth it provided.

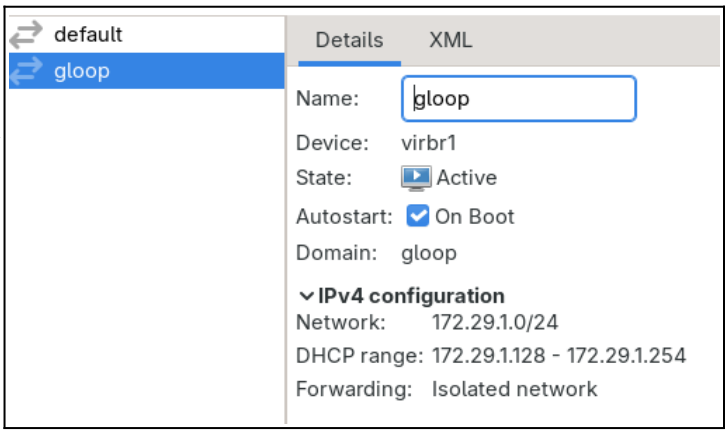


I instead opted to use a fresh install of Arch Linux, with the plan to install all the necessary tools manually. Because of package managers and the Arch User Repository, this would be much more convenient than on Windows. The main tools I installed are: Ghidra and INetSim.

```
user@archlinux /o/i/conf> ls
inetsim.conf
user@archlinux /o/i/conf> pwd
/opt/inetsim/conf
user@archlinux /o/i/conf> which inetsim
/opt/inetsim/inetsim
user@archlinux /o/i/conf>
```

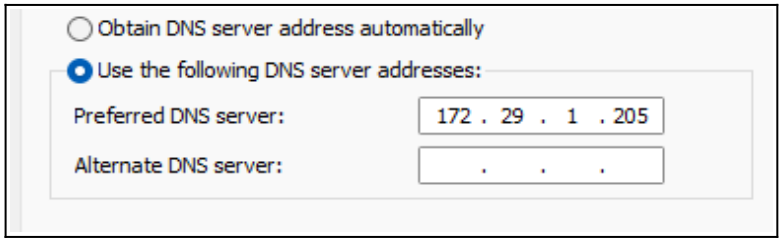
For INetSim in particular, there is an important mention. All online resources I found say that the InetSim config file can be found at /etc/inetsim/inetsim.conf. However, with the AUR install of InetSim, this was not the case: the config file is at /opt/inetsim/conf/inetsim.conf. I only figured this out by reading through the installation logs. The PATH also had to be updated to include /opt/inetsim; since that is where the binary is located.

Network



The final puzzle piece in the lab environment setup is the network. Since I will be messing around with live malware, it is very crucial that it does not have the ability to reach out to the Internet or other devices on the network. For this, I have created a new isolated network which will only be shared between the two virtual machines (victim and analysis). I configured it with an address of “172.29.1.0” and a netmask of “255.255.255.0”. I used the IP range of 172 because the school uses 10 and the default NAT uses 192 — this helps me easily distinguish the networks within the VM based on IP alone.

The main piece of software I really wanted to set up was INetSim. The ability to simulate responses to network requests was too cool not to try. Though, unlike REMnux which already comes with it preinstalled, I had to set it up myself. The installation itself went smoothly, but getting it to work was a whole other story. Configuring the DNS in Windows to use the analysis VM did not yield successful results.



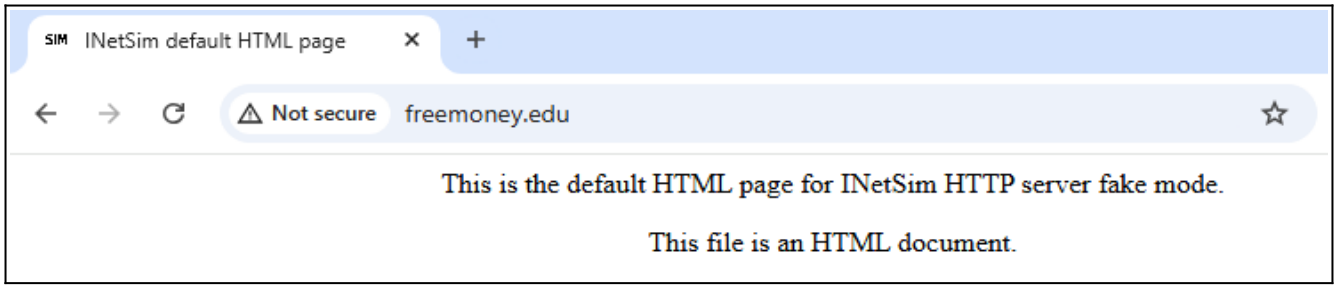

```
cpan
cpan[1]>force get NLNETLABS/Net-DNS-1.37.tar.gz
cpan[2]>install NLNETLABS/Net-DNS-1.37.tar.gz
cpan[3]>exit
```

The fix involved using the cpan program to downgrade the version of Net-DNS to 1.37, which still has the required “main_loop” module.

It took me multiple hours (not an exaggeration) to diagnose and fix the issue. The problem is that the latest version of InetSim (1.3.2) was released in 2020 and relies on a perl “main_loop” module within the Net-DNS library. This loop module no longer exists in the newer versions of the perl-net-dns package which gets installed by the pacman package manager as a dependency of inetsim. Thus, the DNS service never starts, and the Windows VM had no server to resolve domains.

After implementing the fix, I was able to successfully run the inetsim binary on the Linux machine.

```
=== INetSim main process started (PID 2734) ===
Session ID:     2734
Listening on:   172.29.1.205
Real Date/Time: 2025-12-04 18:50:38
Fake Date/Time: 2025-12-04 18:50:38 (Delta: 0 seconds)
 Forking services...
  * dns_53_tcp_udp - started (PID 2736)
  * http_80_tcp - started (PID 2737)
  * ftp_21_tcp - started (PID 2743)
  * smtps_465_tcp - started (PID 2740)
  * smtp_25_tcp - started (PID 2739)
  * https_443_tcp - started (PID 2738)
  * pop3s_995_tcp - started (PID 2742)
  * ftps_990_tcp - started (PID 2744)
  * pop3_110_tcp - started (PID 2741)
 done.
Simulation running.
```




```
FLARE-VM Thu 12/04/2025 18:56:10.30
C:\Users\user\Desktop>nslookup google.com
Server:  www.inetsim.org
Address:  172.29.1.205

Name:    google.com.gloop
Address:  172.29.1.205
```

As a result, all DNS requests made from the Windows victim virtual machine are now being redirected to InetSim on the Linux machine.

I originally decided to forego REMnux VM because it had network issues, but I wonder it would have taken less time to fix that then it did to make InetSim work on Arch Linux. On the other hand, finding solutions to such problems is undoubtedly valuable learning experience.

Malware Analysis: Zeus

Details

Name: ZeusBankingVersion_26Nov2013

Source: https://github.com/ytisf/theZoo/tree/master/malware/Binaries/ZeusBankingVersion_26Nov2013

Executable: invoice_2318362983713_823931342io.pdf.exe

Background: Zeus, originally detected in 2007, is a malware Trojan designed primarily as a bank info-stealer which would inject a key-logger into the victim's browser — capturing online bank credentials. It mainly propagated thorough drive-by downloads (unintended downloads) and phishing. In this case, it attempts to disguise itself as a Portable Document Format (PDF) file. By default, Windows does not have file extensions enabled, so the real ".exe" extension would not be noticed by the average user. Later variants of Zeus also added a botnet component on top of the info-stealer. However, based solely on the name of the current sample, it could very well only have the bank component.

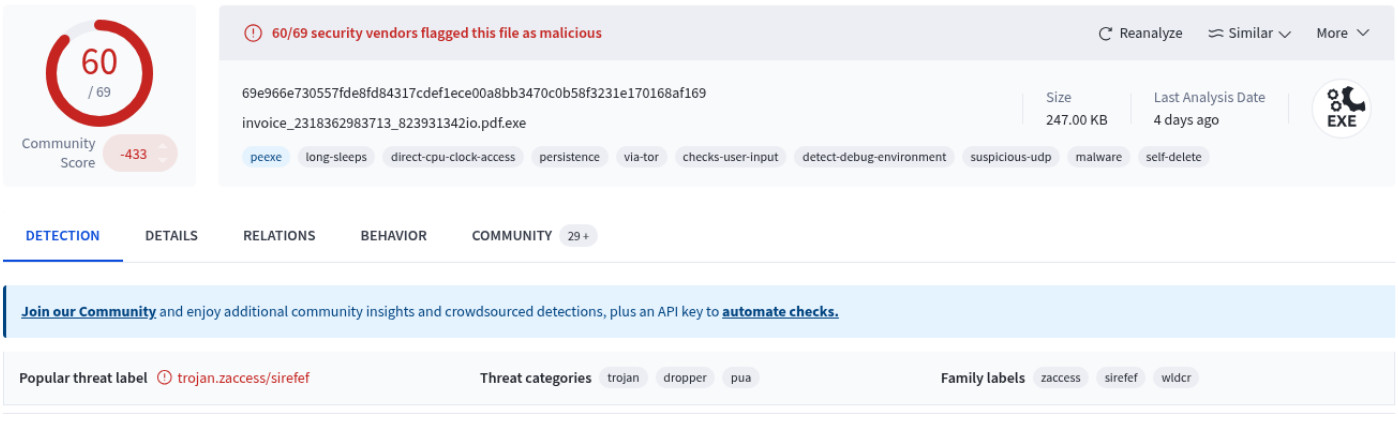
Basic Static

MD5 Hash: ea039a854d20d7734c5add48f1a51c34

SHA1 Hash: 9615dca4c0e46b8a39de5428af7db060399230b2

SHA256 Hash: 69e966e730557fde8fd84317cdef1ece00a8bb3470c0b58f3231e170168af169

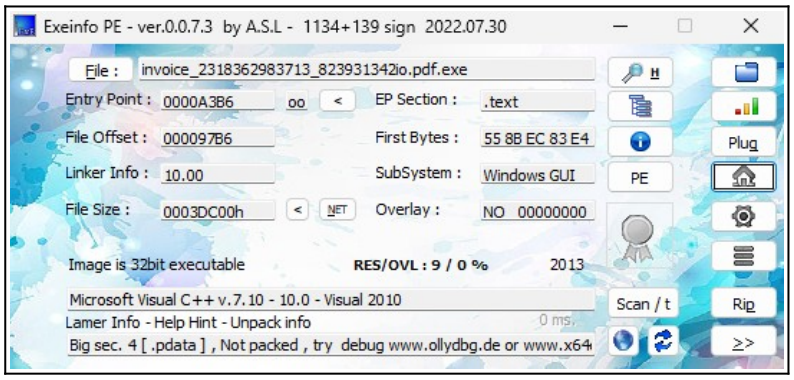
File Info: PE32 executable for MS Windows 5.01 (GUI), Intel i386, 6 sections



Seeing as this Zeus sample is from 2013, it is no supersize that it gets flagged by most of the anti-virus engines on VirusTotal. In fact, I am more surprised by the fact that 9 of the engines determined the binary to be safe. VirusTotal provides other pieces of information which may come in handy further into the analysis. Tags such as direct-cpu-clock-access, persistence, self-delete, and detect-debug-environment provide some insight to the workings of the Zeus Trojan.

Capability	Namespace
reference anti-VM strings targeting VMWare	anti-analysis/anti-vm/vm-detection
encrypt data using chaskey	data-manipulation/encryption/chaskey
resolve function by parsing PE exports (2 matches)	load-code/pe

Using CAPA, a static analysis tool within FLARE-VM for detecting capabilities of binaries, it can be observed that this Zeus malware sample likely has some anti-debugging or anti-disassembly component. Which was also noted previously in the VirusTotal tags. Since the sample is so old, these techniques are unlikely to be sophisticated



Chucking the binary into Exeinfo, I was delighted to see that there is no packer or obfuscator at play. The other Zues samples found in theZoo repository actually do use packers. The lack of packers in this case makes statically analyzing strings and IAT very straightforward.

Speaking of the IAT, pestudio has a convenient flag column which seems to indicate that a particular DLL import is often used for malicious purposes. The Zeus binary has 77 total imports, with 17 being flagged.

Some which stand out are: WinExec, WriteFile and GetClipboardData. GetAsyncKeyState may be the function used for the key-logging.

imports (77)	flag (17)
PathRenameExtensionA	x
WinExec	x
FindNextFileA	x
GetEnvironmentVariableA	x
GetConsoleAliasExesLengthW	x
WriteFile	x
VirtualQueryEx	x
GetCurrentThread	x
GetEnvironmentVariableW	x
GlobalAddAtomA	x
GetClipboardOwner	x
GetClipboardData	x
GetAsyncKeyState	x
EnumClipboardFormats	x
DdeQueryNextServer	x
VkKeyScanA	x
AllowSetForegroundWindow	x

Using the same pestudio tool to now look through the strings, I found something peculiar. There are these pseudorandom strings which look like words. Having random binary data interpreted as strings is normal, however, in this case these are composed solely of alphabetic characters following a PascalCase — which suggests that there could be something else at play.

0x000315A2	-	AsksmaceaglyBubuPulsKaifTeasMistPeelGhisPrimChao
0x000315DC	-	KERNEL32.MulDiv
0x000315EC	-	BagsSpicDollBikeAzonPoopHamsPyasmap
0x00031610	-	KERNEL32.SetCurrentDirectory
0x0003162E	-	BardHolyawe
0x0003163A	-	SHLWAPI.SHFreeShared
0x0003164F	-	BathEftsDawnvilepughThroCymakohloverMitefuzerat
0x0003167F	-	SHLWAPI.PathMakeSystemFolder
0x0003169D	-	BemaCadsPodsWavyCedeRadsbrioOustPerefenom
0x000316C7	-	USER32.SetDlgItemText
0x000316DE	-	BullbonyaweeWaitsnugTierDriblibye
0x00031700	-	KERNEL32.VirtualQuery
0x00031716	-	CameValeWauler
0x00031725	-	USER32.IsIconic
0x00031735	-	CedeSalsshulLimyThroliraValeDonabox
0x00031759	-	USER32.CreateCaret
0x0003176C	-	CellrotoCrudUntohighCols
0x00031785	-	KERNEL32.CreateFile
0x0003179A	-	DenyLubeDunssawsOresvarut
0x000317B4	-	SHLWAPI.PathRemoveFileSpec
0x000317D0	-	DragRoutflusCrowPeatmownNewsyaksSerfmare
0x000317F9	-	USER32.DestroyIcon
0x0003180C	-	Dumpcotsavo

When sorted by offset, DLL function names can be seen below each of these “random” strings.

Just as an assumption, this may be a way to indirectly call these library functions within the Zues binary to make disassembly more difficult. I will try to look into this theory during the advanced static analysis portion of the project.

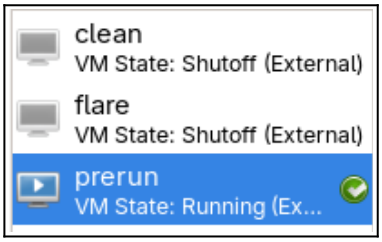
MinorVersion	000a	WORD	0000		
Name	000c	DWORD	**000333f6**	Hex	corect.com
Base	0010	DWORD	**00000001**		
NumberOfFunctions	0014	DWORD	00000000		

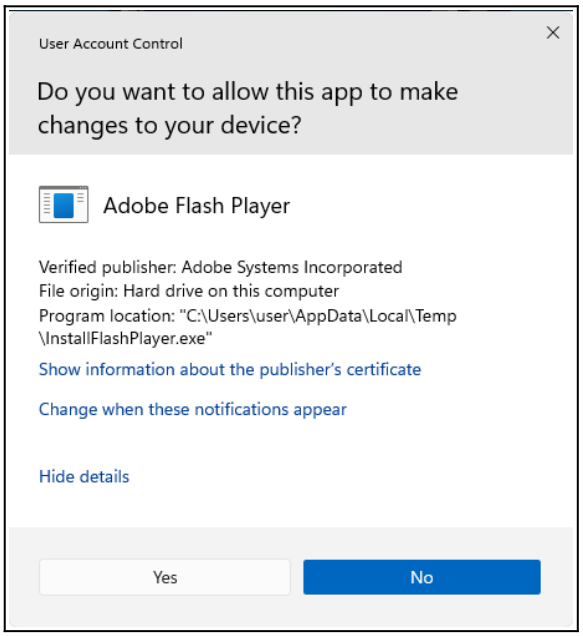
Using the Detect It Easy (die) tool, I also found reference to a “corect.com” url within the exports of the binary.

Going to that url now, it redirected me to a Romanian gambling site. As mentioned previously, a common way for Zeus to infect new victims was through “drive-by downloads” — websites would initiate file downloads without the users knowledge or understanding. It could be that “corect.com” was the source of this particular Zeus strand.

Basic Dynamic

Right before running the Zeus binary, I took a snapshot of the system in a running state. This will make rerunning the malware as simple as going back to the snapshot — with everything already set-up the way I need it.

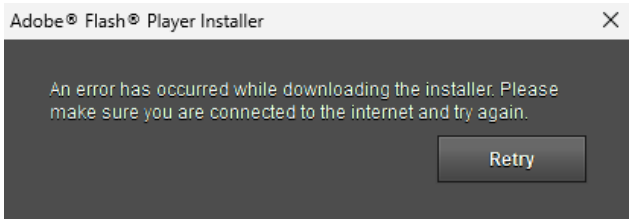




With INetSim up and running on the Linux machine, I wanted to start of by observing what happens when running invoice_2318362983713_823931342io.pdf.exe without any forensic tools.

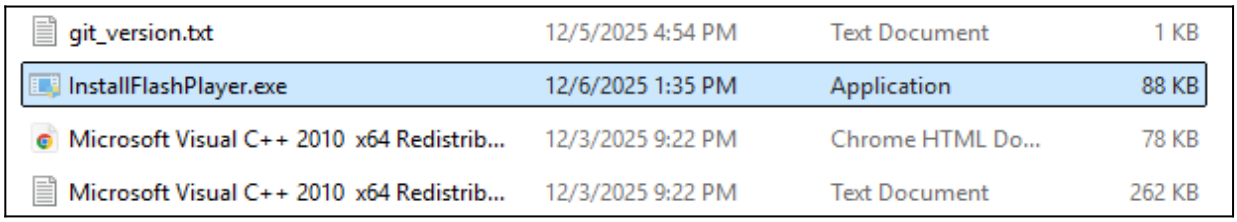
Double clicking the executable, nothing appears to happen for a good 10-15 seconds, after which a popup shows up. This popup is part of User Account Control, indicating that the "Adobe Flash Player" wants administrator level permissions.

Clicking "No" simply re-triggers the popup, meaning the only way to proceed is to click "Yes". After doing so, another popup appears this time saying there was an error during the download process.

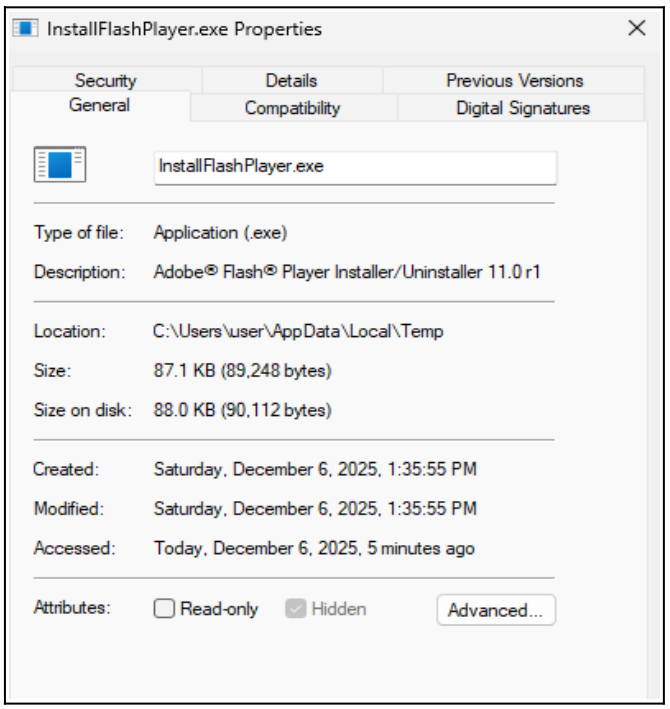


Right now, it is hard to say whether this is the intended behavior or if it comes as a result of not having a real Internet connection.

After this error popup, the executed Zeus binary is deleted.

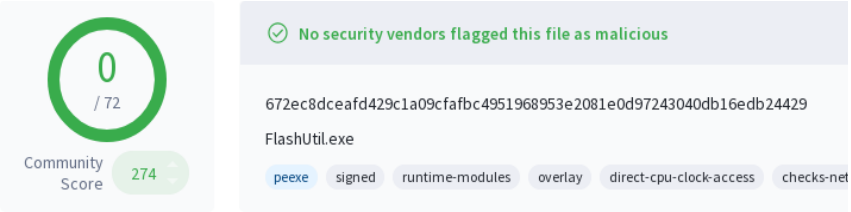


Navigating to “C:\Users\user\AppData\Local\Temp” as shown in the UAC popup, I was able to locate “InstallFlashPlayer.exe”. It is marked hidden.

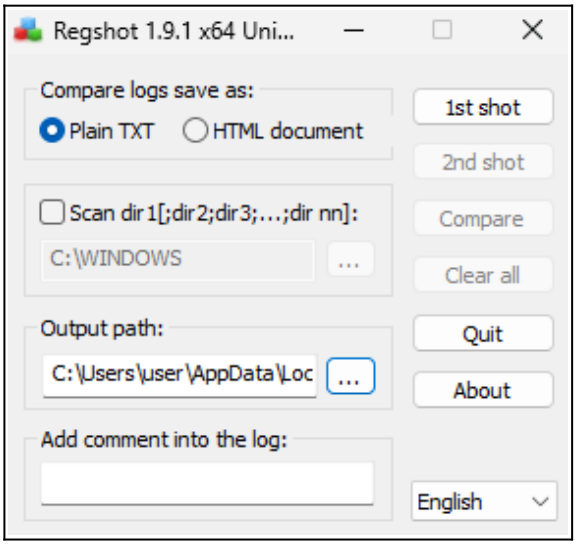


Looking though the signatures, this appears to be a legitimate Adobe installation binary. Uploading the file hash onto VirusTotal, shows no detections. It is possible that Zeus hooks into this installer to hide it’s operations behind an application which most people would not think twice about.

By going back to the previous snapshot, I confirmed that InstallFlashPlayer.exe was not at that location before Zeus runs.



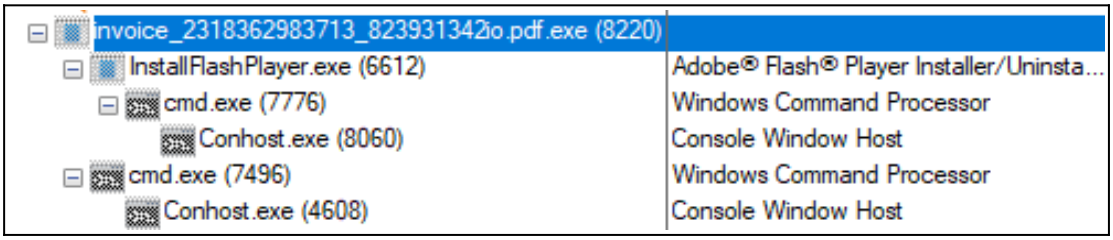
The next stage involved using tools such as Procmon, Regshot, and Wireshark to observe what Zues was doing behind the scenes.



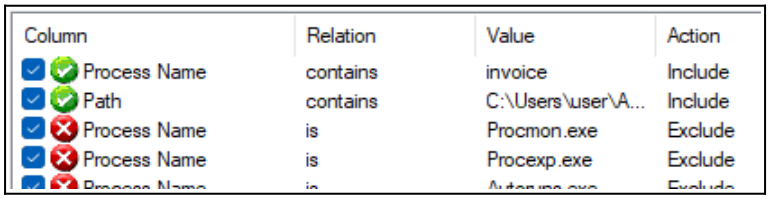
Using Regshot: I took the first shot right before executing the malware, and the second shot right after the AdobeInstaller error popup. Despite such close time proximity, the comparison file had quite a significant amount of data — which for someone as inexperienced as me did not make things easy.

I did find multiple references to “InstallFlashPlayer.exe”, but beyond that nothing in particular stood out. I will try to take a look at the registry modifications again a bit later, using Procmon, since I believe it allows filtering activity by application.

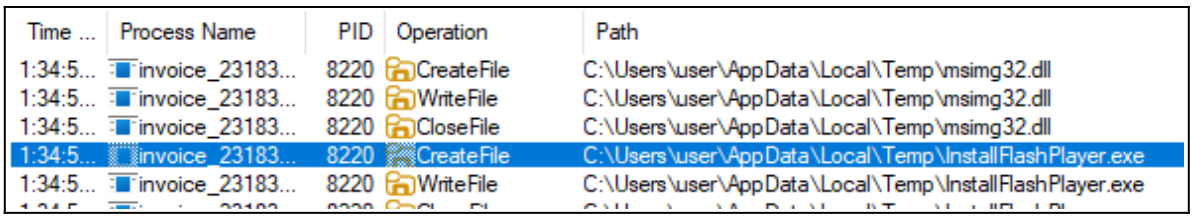
Looking though the Procmon process tree, it can be observed that the initial invoice executable initiates the “InstallFlashPlayer.exe” process. So far this is inline with what I observed earlier. The Flash installed then proceeds to launch a Windows command process.



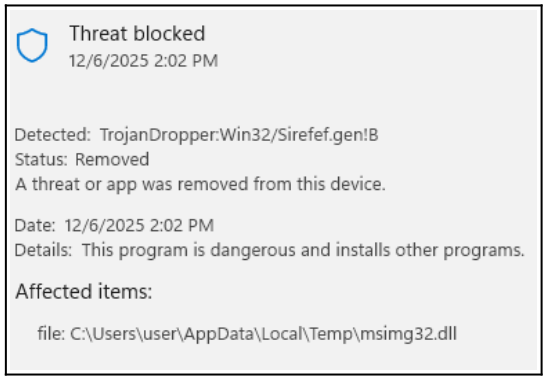
The default process monitor window inside of Procmon contains far too much activity (almost 800k events) to go though as is. So I applied a few filters to narrow down the search.



The main filter ensures that all events are made by the malware sample (which starts with “invoice”). The second filter looks at activity involving the “C:\Users\user\AppData\Local\Temp” directory, in which I know the Flash installer is placed by the malware.

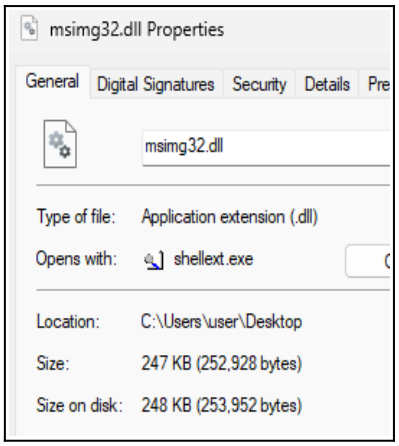


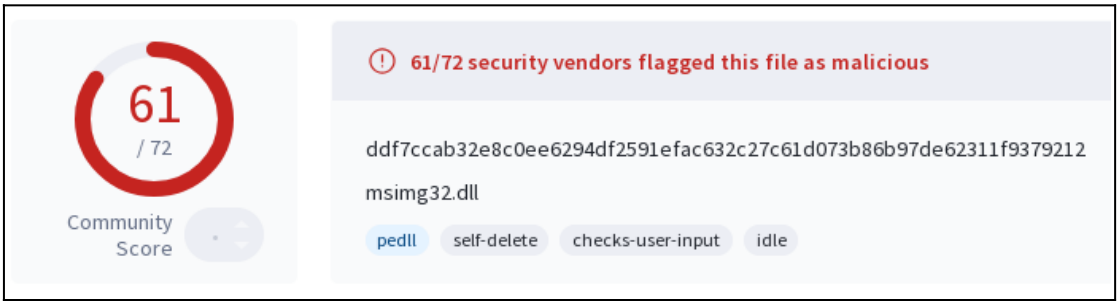
As expected, the logs show that the Zeus sample creates the “InstallFlashPlayer.exe” file. However just before that, another file is created in the same directory: “msimg32.dll”.



I initially could not locate this dll file and assumed that the malware was deleting it after execution. However, looking though Windows logs, I realized that it was actually being deleted by Windows, despite me having turned off Defender through group policy.

I supposed taking control away from users is par for the course with Microsoft. Anyway, after struggling with Windows Defender for a while, I was able to get my hands on the “msimg32.dll” file by taking a copy before it got deleted.



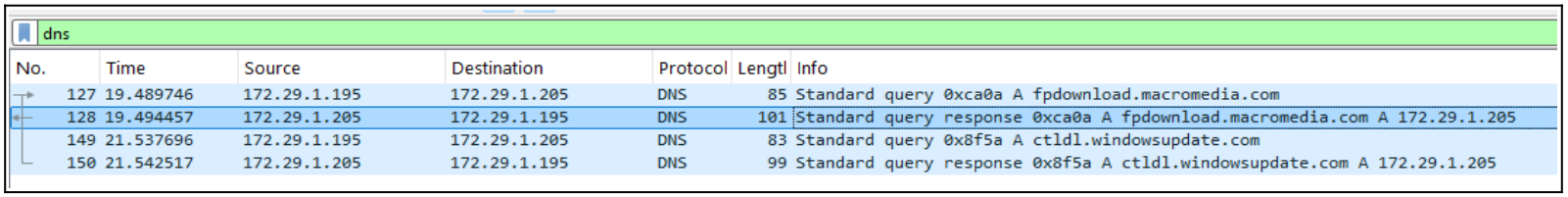


Searching VirusTotal with the hash of the file, it becomes quite obvious that this is a malicious DLL. In fact, based on some of the tags, it could be that this is the actual key-logging component of the malware

Although this is the dynamic analysis portion of the project, the fact that getting the DLL requires running the malware, compelled me to quickly look at it’s disassembly using IDA. A quick glace made it obvious that despite having the “.dll” extension, this is the same binary as “invoice_2318362983713_823931342io.pdf.exe”. It is highly likely that the purpose of the DLL is for the malware to establish persistence.

To further this idea of persistence, it appears that invoice malware process sets a registry value at HKCU\Software\Microsoft\Windows\CurrentVersion\Run. This registry location specifies programs to run automatically each time a user logs into the system. So, it seems Zeus is using “Google Update” as a way to launch the key-logging DLL payload each time a user signs in.

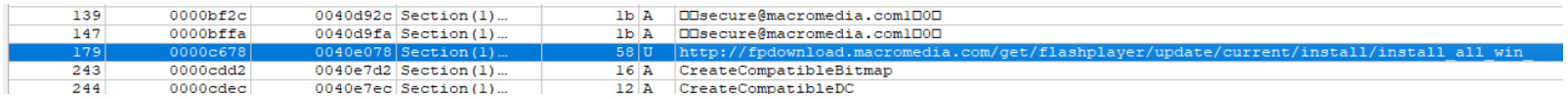
The final thing I wanted to check was the network traffic. Using Wireshark I took note of a DNS request made by the malware: “fpdownload.macromedia.com”.



dns

No.	Time	Source	Destination	Protocol	Lengtl	Info
127	19.489746	172.29.1.195	172.29.1.205	DNS	85	Standard query 0xca0a A fpdownload.macromedia.com
128	19.494457	172.29.1.205	172.29.1.195	DNS	101	Standard query response 0xca0a A fpdownload.macromedia.com A 172.29.1.205
149	21.537696	172.29.1.195	172.29.1.205	DNS	83	Standard query 0x8f5a A ctldl.windowsupdate.com
150	21.542517	172.29.1.205	172.29.1.195	DNS	99	Standard query response 0x8f5a A ctldl.windowsupdate.com A 172.29.1.205

Since this malware is meant to extract data from the victim’s device, this website could be the attacker’s server. However, this particular URL is hard-coded into the “InstallFlashPlayer.exe” which I previously determined to be a legitimate process.



139	0000bf2c	0040d92c	Section(1)…	1b	A	□□secure@macromedia.com1□0□
147	0000bffa	0040d9fa	Section(1)…	1b	A	□□secure@macromedia.com1□0□
179	0000c678	0040e078	Section(1)…	58	U	http://fpdownload.macromedia.com/get/flashplayer/update/current/install/install_all_win_
243	0000cdd2	0040e7d2	Section(1)…	16	A	CreateCompatibleBitmap
244	0000cdec	0040e7ec	Section(1)…	12	A	CreateCompatibleDC

Aside from this, there were no other suspicious network requests made. This probably means that the malware does not attempt to establish immediate contact with the hackers. Alternatively, it may have detected that it was executed inside a virtual environment and adjusted it’s behavior accordingly.

Advanced Static

Disclaimer: This malware is much more sophisticated than anything I have experience disassembling for class assignments. Seeing as I am the only person working on this project, I simply have no time to offer a full disassembly writeup similar in scope to the previous two sections. That being said, I will do my best to analyze a few interesting parts within the Zeus binary.

I used a combination of Ghidra and IDA to perform the advanced static analysis. This mostly comes down to my desire to experiment with different disassemblers.

```
0040a4be be 00 00 08 00     MOV         ESI,0x80000

                        LAB_0040a4c3                                    XREF[1]:
0040a4c3 ff 15 4c 00 42 00  CALL        dword ptr [->KERNEL32.DLL::GetTickCount]
0040a4c9 4e                 DEC         ESI
0040a4ca 75 f7              JNZ         LAB_0040a4c3
```

Loading the Zeus binary into Ghidra, the first thing that jumped out was a call to the "GetTickCount" function within the KERNEL32 DLL. This function is used to retrieve the number of milliseconds that have elapsed since the system was started. My initial assumption was that this could be a way for the malware to detect if it was operating inside a virtual environment — since most users don't often shutdown their devices. However, as is fairly evident, the function's results are never accessed, nor saved for that matter. Instead, this is a while-loop which just calls "GetTickCount" 524288 times. This would explain the 10-15 second delay when launching the malware. I have no real ideas as to why an artificial delay such as this would be implemented into malware. My only guess would be that it may bypass some anti-virus software that only checks a processes runtime for the first few seconds to determine if it is a malware.

```
0040a552 6a 52              PUSH        0x52
0040a554 6a 44              PUSH        0x44
0040a556 d3 e8              SHR         EAX,CL
0040a558 8b 4c 24 1c        MOV         ECX,dword ptr [ESP + 0x1c]
0040a55c 8d 44 01 e5        LEA         EAX,[ECX + EAX*0x1 + -0x1b]
0040a560 89 44 24 1c        MOV         dword ptr [ESP + 0x1c],EAX
0040a564 8b 44 24 1c        MOV         EAX,dword ptr [ESP + 0x1c]
0040a568 ff d0              CALL        EAX
0040a56a a1 e0 fc           MOV         EAX,[DAT_0040fce0]
```

I also took note of the fact that the malware rarely has direct calls to functions. It would instead manipulate and later call the value inside the EAX register. This this is such a common occurrence within the binary, that I assume it's an anti-disassembly technique employed to make my life harder.

This is honestly a fairly effective obfuscation strategy, especially with how many different local variables there are in the entry point function alone. It would undoubtedly take me a significant amount of time to decode all these indirect function calls.

Based on prior static analysis, I knew that there were a number of DLL functions within the IAT that could be used maliciously. Using Ghidra, I was able to locate them within the binary.

```
                    LAB_0040b6e4
0040b6e4 f6 05 98        TEST        byte ptr [DAT_00410b98],0x10
         0b 41 00 10
0040b6eb 75 11           JNZ         LAB_0040b6fe
0040b6ed a1 08 01        MOV         EAX,[->USER32.DLL::GetCapture]
         42 00
0040b6f2 83 0d 98        OR          dword ptr [DAT_00410b98],0x10
         0b 41 00 10
0040b6f9 a3 2c 07        MOV         [DAT_0041072c],EAX
         41 00
```

Here, a call to “GetCapture” is used to determine which window is currently capturing user mouse input. This is likely used by Zeus to determine when a browser is used.

I also found a call to “GetAsyncKeyState”. This library function is often utilized by malware to capture user keystrokes in real time.

```
00404d05 66 3b c1        CMP         AX,CX
00404d08 0f 84 76        JZ          LAB_00405484
         07 00 00
00404d0e a1 fc 00        MOV         EAX,[->USER32.DLL::GetAsyncKeyState]
         42 00
00404d13 89 45 fc        MOV         dword ptr [EBP + -0x4],EAX
00404d16 89 75 dc        MOV         dword ptr [EBP + -0x24],ESI
00404d19 a1 00 f0        MOV         EAX,[DAT_0040f000]
         40 00
```

In both cases, the usage of these DLL functions is nested deeply inside other sub-functions, so gaging the full extent of their application is challenging. However, since I know that this Zeus variant is primarily a bank info-stealer, I think it’s fair to assume that these functions are used in conjunction to enable key-logging each time the user opens their browser.

```
                    ****************************************************************
                    *                          FUNCTION                            *
                    ****************************************************************
                    int __stdcall FUN_0040bb92(void)
                      assume FS_OFFSET = 0xffdff000
    int               EAX:4          <RETURN>
                    FUN_0040bb92                                XREF[1]:     FUN
0040bb92 a1 24 08        MOV         EAX,[DAT_00410824]
         41 00
0040bb97 35 0f 16        XOR         EAX,0x160f
         00 00
0040bb9c 2d c4 4c        SUB         EAX,0x4cc4
         00 00
0040bba1 c2 18 00        RET         0x18
```

The last thing I wanted to point out, is the sheer number of functions with very basic operations. In a regular program there may be a few small helper functions, but this is on a whole different scale. I would approximate that over 80% of functions in this Zeus binary are one-liner calculations performed on some data. This very much seems like more anti-disassembly/obfuscation techniques.

Conclusion

Summing both the static and dynamic analysis findings together, the general outline of how Zeus operates can be summarized with the following diagram: