VB2020
localhost

30 September - 2 October, 2020 / vblocalhost.com

LIGHTWEIGHT EMULATION BASED IOC EXTRACTION FOR GAFGYT BOTNETS

Ya Liu

Qihoo 360 Technology, China

liuya@360.cn

ABSTRACT

The long-established botnet family of Gafgyt is still very active today. While new C2 servers emerge quickly, they usually remain active for only a few days. To effectively fight such quickly emerging while short-lived families, quick IoC extraction is import. In this paper I will introduce how to extract IoCs for Gafgyt by exploiting its characteristic C2 loop and by lightweight emulating its initConnection function. The introduced methods can also be used for variant classification and tracking. To better understand how Mirai code is used in Gafgyt, some widely used Mirai code is also investigated.

1. INTRODUCTION

Gafgyt, also known as BASHLITE and Qbot [1], was designed to infect *Linux* devices to launch DDoS attacks, with the original version found in 2014. In recent years, we have observed the proliferation of Gafgyt variants. While that proliferation can mainly be explained by the fact that the Gafgyt source code was leaked in 2015, the success of Mirai, together with its leaked source and tens of thousands of off-the-shelf vulnerable IoT devices, might also have contributed a lot to that.

On the other hand, Gafgyt botnets are usually short lived. According to our data, most of the tracked C2 servers only remained active for a few days. Therefore, quick IoC extraction would play an important role in fighting against such quickly emerging while short-lived botnets. In this paper the IoCs refer to two kinds of information: 1) the C2 server and port; 2) the register message. While in early variants the C2 server and port are usually stored in strings, later variants usually binary encode them, which makes them more difficult to extract. As for the register message, this is the first message a bot sends to its C2 server after establishing a connection. Sometimes it's also referred to as check-in, call-home or HELLO. With the register message, we can: 1) define an IDS/IPS rule to distinguish Gafgyt communication from real network traffic; 2) track its variants.

For the extraction of IoCs, we first turned to the use of a sandbox. While in most cases this works, there exist issues of evasion, deployment, long run time, and security risks such as network scanning. The second solution is static analysis based. It works as follows:

1. IoC related code snippets are located using their signatures, e.g. YARA.
2. The relevant instructions are parsed to get the wanted data.

This solution relies heavily on static code signatures. Since Gafgyt targets multiple processor architectures, when a new variant emerges, different signatures for each CPU architecture have to be defined, which is both tedious and time consuming, and may also lead to the signature explosion issue.

On the other hand, fixed patterns, both static and dynamic, exist in Gafgyt's C2 communication-related code. More specifically, there is a characteristic C2 communication code loop in Gafgyt that can be recognized using its static patterns. With the C2 loop, both the IoC of the register message and the function needed for lightweight emulation to extract C2 information can be directly checked. That finding inspired me to develop the hybrid solution that will be introduced in this paper. Basically, it works in two steps: 1) recognizing the C2 loop with its CFG patterns; 2) lightweight emulating the target function to extract the C2 server and port. Compared with the sandbox solution, it's easy to deploy, and since only a subset of code is executed, the runtime is greatly reduced. Compared with a purely static analysis-based solution, the signature explosion issue is eliminated since only a relatively small number of behaviour patterns are needed.

The remainder of this paper is organized as follows: in Section 2, I introduce the C2 loop, including its patterns and recognition; in Section 3, I introduce the behaviour patterns in initConnection and how to use them to lightweight emulate initConnection to extract the C2 server and port; in Section 4 I investigate some widely used Mirai code in Gafgyt with Mirai's characteristic encrypted configurations.

To summarize, the contributions of this paper are as follows:

- I summarize the fixed patterns in the Gafgyt C2 loop and behaviour patterns in connection establishment that can be used for IoC extraction and variant classification.
- I demonstrate a solution for automatically extracting C2 information by lightweight emulating a specific function with its behaviour patterns.

Since Gafgyt targets multiple processor architectures, the same source code is usually compiled into multiple binaries. For reasons of simplicity and efficiency, only samples for x86, x64, MIPS and ARM are considered.

The SHA256 hashes for the samples discussed in this paper are given in Appendix A.

2. THE C2 LOOP AND CONNECTION ESTABLISHMENT

In DDoS purposed botnets, the C2 communications including establishing connection, registering with the C2 server, and receiving and responding commands, are usually embedded in a code loop which is called the C2 loop in this paper. When analysing a new botnet family or unknown sample, the C2 loop is a good entry for reverse engineering its C2 protocol and extracting IoCs. Furthermore, C2 loops are usually specific to their families, thus can be used for family recognition and variant classification.

In Gafgyt, the C2 loop is in main(), with connection establishing, registering, and command receiving done in three functions. In the leaked source and some unstripped samples, as shown in Figure 1, those three functions are separately named as initConnection, sockprintf, and recvLine. In some variants they might have different names. For example, sockprintf is also known as botnetPrint, HackerPrint, HeliosPrint and socketSend. For the convenience of describing, I will use initConnection, sockprintf, and recvLine to refer to them.

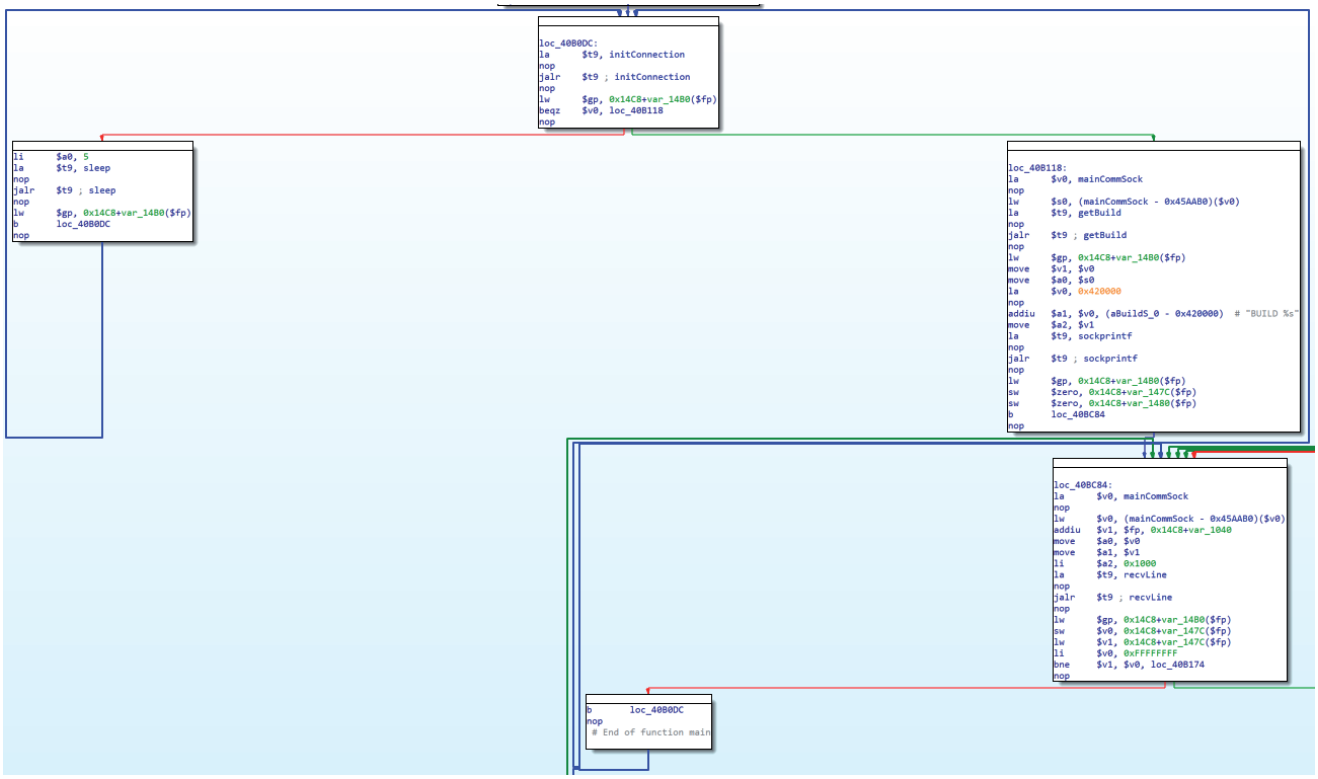

Figure 1: Gafgyt C2 loop (MD5= 5264b21d93ce4668c3f8aac823924c04).

For simplicity, a text format of loop description is introduced. The C2 loop in Figure 1 can be represented as follows:

```
"[initConnection][] -> [getBuild, sockprintf]["BUILD %s"] -> [recvLine][] -> [][]"
```

Code blocks are connected with '->'. Since we only care about the called functions and referenced strings, each block is represented with two '[]', with the first '[]' enclosing the called functions while the second one encloses the referenced strings.

With the C2 loop, both the register message and the initConnection function can be directly obtained. The register message in Figure 1 corresponds to 'BUILD %s'. And the initConnection function can be taken from the first block for further emulation to extract the C2 server and port. However, as new variants continuously emerge, C2 loops also vary in forms, as shown by the following lines:

```
"[initConnection][] -> [jprintf]["arch %s", "unknown"] -> [recvLine][] -> [][]"
"[initConnection][] -> [][] -> [recvLine][] -> [][]"
"[echoconnection][] -> [][] -> [recvLine][] -> [][]"
"[initConnection][] -> [sprintf, sockprintf]["fftt:%s"] -> [recvLine][] -> [][]"
"[Connection, botkiller, recv_buf][] "
```

For better loop detection, common C2 loops are summarized into six types based on their CFG patterns. For each type, a set of criteria are defined in terms of block number, called functions, and referenced strings. For example, the C2 loop shown in Figure 1 belongs to type 1. Its criteria are as follows:

1. The loop is composed of four blocks.
2. Only one function is called in the first block, which corresponds to initConnection.
3. At least two functions are called in the second block.
4. At least one string containing '%s' is referenced in the second block.
5. Only one function is called in the third block.

All six types of C2 loops, together with their criteria, are illustrated in Appendix B. Their coverage stats on all 116,677 samples are illustrated in Table 1.

Type	Samples
1	93,140
2	4,344
3	17,418
4	801
5	333
6	641

Table 1: C2 loop stats.

C2 loop detection is done in the static analysis stage. It includes two steps: 1) finding all loops in main(); 2) checking each loop type with the criteria introduced in Appendix B. The static analysis can be achieved with common scriptable reverse engineering tools, e.g. *IDA*, *radare2*. In my solution, *radare2* is used.

In step 2, CFG patterns alone may not be enough to determine the loop type because it's common for similar loops to exist in main(). The register message and characteristic of initConnection are used to solve that problem due to their tight connections with the C2 loop. While some widely used format strings such as 'BUILD %s', 'fttt %s', 'arch %s' can be directly used to check the register message, the fact that '%s' always exists in the format string is used to heuristically detect other unknown register messages.

Gafgyt is also characteristic in its initConnection function in terms of CFG patterns and dynamic behaviours. More on that will be introduced in Section 3. For simplicity, the criteria for determining initConnection are as follows:

1. block_number >= 3 and block_number <= 16.
2. edge_number >= block_number and edge_number <= block_number + 5.
3. called_functions >= 2.
4. referenced_string_number > 0.

The above criteria must be used together with the target C2 loop CFG patterns. As an example, consider the second sample of type 2 illustrated in Appendix B. In total, 16 loops can be found in main(), as shown in Figure 2. In step 2, each loop will be checked with the illustrated type in turn. When it comes to the loop of '[fcn.0804d14f][] -> [][] -> [fcn.08048a8e][] -> [] []', only after the loop CFG patterns successfully match type 2 would the function of fcn.0804d14f in the first block be matched with the above initConnection criteria.

```
"[fcn.0804d14f][] -> [][] -> [fcn.08048a8e][] -> [][]"
"[][] -> [fcn.0804db3c][] -> [][] -> [][] -> [fcn.0804e3f2][] -> [][] -> [][] -> [fcn.0804edc9][] -> [][]"
"[][] -> [fcn.0804db3c][] -> [][]"
"[fcn.08048a8e][] -> [][] -> [][] -> [][] -> [][] -> [fcn.080482bb, fcn.0804dc20]["$E$rQ"] -> [fcn.080488ba]["$t$<$9E$KI"]"
"[fcn.08048a8e][] -> [][] -> [][] -> [][] -> [][] -> [fcn.080482bb, fcn.0804dc20]["$E$rQ"] -> [fcn.0804dc20]["<$9E$KI"] -> [fcn.0804f574][] -> [][]"
"[][] -> [][] -> [fcn.0804e3f2, fcn.0804dc04][] -> [fcn.0804dd08]["9E$KI"]"
"[][] -> [][] -> [fcn.0804dd08]["9E$KI"]"
"[][] -> [fcn.0804edc9][]"
"[][] -> [fcn.0804dba4][]"
"[fcn.08048a8e][] -> [][] -> [][] -> [][] -> [][] -> [fcn.080482bb, fcn.0804dc20]["$E$rQ"] -> [fcn.0804dc20]["<$9E$KI"] -> [][]"
"[fcn.08048a8e][] -> [][] -> [][] -> [][] -> [fcn.080482bb, fcn.0804dc20]["$E$rQ"] -> [fcn.080488ba]["$t$<$9E$KI"]"
"[fcn.08048a8e][] -> [][] -> [][] -> [][] -> [fcn.080482bb, fcn.0804dc20]["$E$rQ"] -> [fcn.0804dc20]["<$9E$KI"] -> [fcn.0804f574][] -> [][]"
"[fcn.08048a8e][] -> [][] -> [][] -> [][] -> [fcn.080482bb, fcn.0804dc20]["$E$rQ"] -> [fcn.0804dc20]["<$9E$KI"] -> [][]"
"[fcn.08048a8e][] -> [][] -> [][] -> [][] -> [fcn.080482bb, fcn.0804dc20]["$E$rQ"] -> [fcn.0804dc20]["<$9E$KI"] -> [][] -> [][] -> [][] -> [][]"
"[fcn.0804d14f][] -> [fcn.0804f5dc][]"
```

Figure 2: Similar loops exist in main() (MD5= 0967a1ad0056ca664e064a59e9f263e1).

3. LIGHTWEIGHT EMULATING INITCONNECTION

Basically, lightweight emulation (LWE for short) is a kind of dynamic analysis technique. It has a long history of being used in detecting shellcode in network data [2]. Different from shellcode detection, a code snippet from an executable file, or the initConnection function in this paper, becomes the emulation target. The emphasis here changes from detecting suspicious behaviours, e.g. locating a system API or loading a system DLL, to making sure the relevant behaviours are properly executed. While it looks a bit similar to sandbox-based dynamic analysis, the difference lies in the fact that in LWE only a subset of code needs to be executed, and there are usually a very limited number of system services provided, or even none at all. For these reasons, the issues of external code/data dependency usually exist in LWE. Except that instruction-level analysis is a MUST to detect the relevant behaviours.

In this paper, the aim of the LWE is to extract the C2 server and port by emulating the initConnection function checked with the C2 loop. The solution can be divided into three stages: pre-handling, emulation, and post analysis. In pre-handling, the function is inspected at instruction level to replace all function calls with NOPs. The purpose of that is to remove external code dependency. In the meantime, the calling addresses are saved for function call checking during emulation.

In stage 2, the unicorn open-source emulation engine is used. Two hooks are installed to detect function calls and memory writes. The first is named 'SingleStep'. It is installed with the UC_HOOK_CODE API, thus will get called every time an instruction is to be executed. It's responsible for two tasks:

1. When detecting the PC of NOP'ed call instructions, SingleStep will generate a CALL event together with a pre-set number of parameters.
2. When detecting the PC of code ending address or an address beyond the initConnection range, SingleStep will stop the emulation.

The second hook is named 'HookWrite'. It is installed with the UC_HOOK_MEM_WRITE API. When a memory write is detected, HookWrite will be called. It will log the WRITE event together with the write address, size and value.

After emulation finishes, the recorded events and the final memory snapshot will be handed to post analysis for C2 extraction. The event formats for exchanging behaviour information between stages 2 and 3 are shown in Figure 4. For simplicity, a fixed number, defaulted to six in my solution, of parameters are recorded for each function call. For a memory write event, the enclosed parameters stand separately for write address, size and value.

The real extraction is based on the initConnection function's behaviour patterns. They are defined by the called functions and memory writes. A behaviour pattern has two layers of meaning: pattern matching and rule applying. When a function's behaviours match a specific type of initConnection behaviour pattern, that pattern's rules will be applied on the behaviours to have data extracted. As an example, consider the unstripped version of initConnection shown in Figure 3. The C2 IP and port of '198.134.120.150:23' are stored in a global variable named *commServer*. After they get parsed by calling *strcpy*, *strchr*, and *atol*, a C2 socket is created and *connectTimeout* is called to initialize the real connection. Since the function call to *strchr* is NOP'ed with value 0 returned, the block where *atol* is called will not be emulated. The final behaviours are shown in Figure 4. For simplicity, only four parameters are shown for each call.

```
_BOOL4 initConnection()
{
  int v0; // eax
  char server[4]; // [esp+14h] [ebp-1004h]
  int port; // [esp+1014h] [ebp-4h]

  memset(server, 0, 4096);
  if ( mainCommSock )
  {
    close(mainCommSock);
    mainCommSock = 0;
  }
  if ( currentServer )
    ++currentServer;
  else
    currentServer = 0;
  strcpy(server, (&commServer)[currentServer]);
  port = (int)&Server_Botport;
  if ( strchr(server, 58) )
  {
    v0 = strchr(server, 58);
    port = atol(v0 + 1);
    *(_BYTE *)strchr(server, 58) = 0;
  }
  mainCommSock = socket(2, 1, 0);
  return connectTimeout(mainCommSock, server, port, 30) == 0;
}
```

Figure 3: An unstripped version of initConnection (MD5=00432f33fb3f5cc5377266a5439567bf).

```
ops in initConnection() (addr=0x0804d94f):
"c", pc=0x0804d971, (0x0124eff8, 0x00000000, 0x00001000, 0x00000000)
"w", pc=0x0804d9b3, (0x0805b4f0, 0x00000004, 0x00000000, 0x-00000001)
"c", pc=0x0804d9d4, (0x0124eff8, 0x080557c0, 0x10101010, 0x10101010)
"c", pc=0x0804d9ef, (0x0124eff8, 0x0000003a, 0x10101010, 0x10101010)
"c", pc=0x0804da3f, (0x00000002, 0x00000001, 0x00000000, 0x10101010)
"w", pc=0x0804da42, (0x0805f3e0, 0x00000004, 0x00000000, 0x-00000001)
"c", pc=0x0804da5f, (0x00000000, 0x0124eff8, 0x0805b4e4, 0x0000001e)
```

Figure 4: Recorded behaviours.

Two patterns can be concluded for behaviours in Figure 4: 'cw4cccw4c' and 'call_memset, w4, call_strcpy, call_strchr, call_socket, w4, call_connectTimeout'. The first pattern is a simplified version for fast matching. Only those successfully matched with 'cw4cccw4c' will be checked with the second one, where every function can be heuristically determined with its characteristic parameters as follows:

1. For *memset*, arg1 points to stack memory while arg2 holding 0 and arg3 usually hold a const of 0x1000.
2. For *strcpy*, arg1 and arg2 point separately to stack and global memory.
3. For *strchr*, arg1 points to stack memory while arg2 holding 0x3a, which stands for ':'.
4. For *socket*, there are always parameters of (2, 1, 0), or (2, 2, 0) in the case of MIPS CPU.
5. For *connectTimeout*, arg2 is equivalent to arg1 of *strcpy* and *strchr*.

If successfully matched, the C2 IP and port can be retrieved from global memory pointed to by *strcpy's* arg2, which is 0x080557c0 in Figure 4.

Similar to the C2 loop, different versions of initConnection functions exist. For better data extraction, they are summarized into six types in terms of behaviour patterns, as illustrated in Appendix C. Except for data extraction, the concluded types can also be used for variant tracking. For example, the initConnection shown in Figure 3 has a slightly mutated version, as shown in Figure 5.

The only difference is that in Figure 3 the C2 IP is read from global memory, while in Figure 5 it is dynamically generated by calling sprintf() with a format of '%d.%d.%d.%d'. They share the same CFG patterns and simplified behaviour patterns. In Appendix C, the initConnection functions in Figure 3 and 7 separately belong to types 1 and 2. Those sorts of similarities can also be found between types 4 and 5, where htonl() is called in type 4 for four-byte binary encoded C2 IP, while inet_addr() for string format C2 IP.

```
_BOOL8 initConnection()
{
  __int64 v0; // rax
  char server; // [rsp+10h] [rbp-1010h]
  int port; // [rsp+101Ch] [rbp-4h]

  memset(&server, 0LL, 4096LL);
  if ( KHcommSOCK )
  {
    close((unsigned int)KHcommSOCK);
    KHcommSOCK = 0;
  }
  if ( KHserverHACKER == 3 )
    KHserverHACKER = 0;
  else
    ++KHserverHACKER;
  szprintf(
    &server,
    "%d.%d.%d.%d",
    (unsigned int)hacks[KHserverHACKER],
    (unsigned int)hacks2[KHserverHACKER],
    (unsigned int)hacks3[KHserverHACKER],
    (unsigned int)hacks4[KHserverHACKER]);
  port = hakai_bp;
  if ( strchr(&server, 58LL) )
  {
    v0 = strchr(&server, 58LL);
```

Figure 5: A slightly mutated version of initConnection in Figure 3 (MD5=001618368ffd8735837267d9763b0fa1).

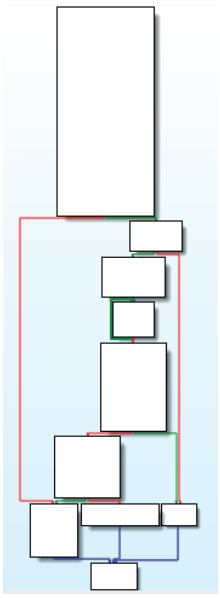

Figure 6: A type 6 initConnection which has complex CFG (MD5= 3cda17370a9c72120001c44fba76b442).

As mentioned above, LWE in this paper only makes sure relevant operations get emulated. That is for consideration of simplicity. As an example, consider the type 6 initConnection shown in Figure 6. Its CFG is so complex that it's difficult to make sure every block get emulated.

However, further studies show the relevant operations are all located in the first block, as shown in Figure 7. What we need to do is to make sure the first block is properly emulated, which will greatly simplify things.

```
PUSH    {R4-R7,LR}
LDR     R6, =comsocket
SUB     SP, SP, #0xAC
MOV     R1, #3
MOV     R2, #0
LDR     R0, [R6]
BL      fcntl
MOV     R1, #4
ORR     R2, R0, #0x800
LDR     R0, [R6]
BL      fcntl
MOV     R1, #1
MOV     R2, #0
MOV     R0, #2
BL      socket
MOV     R3, #0x2327
STR     R0, [R6]
STRH    R3, [SP,#0xC0+var_36]
LDR     R0, =a167_71_33_152 ; "167.71.33.152"
MOV     R3, #2
STRH    R3, [SP,#0xC0+var_38]
BL      inet_addr
ADD     R1, SP, #0xC0+var_38
STR     R0, [SP,#0xC0+var_34]
MOV     R2, #0x10
LDR     R0, [R6]
BL      connect
CMP     R0, #0
BLT     loc_84B4
```

Figure 7: The first block of the initConnection function in Figure 6.

For those cases where relevant operations are deep inside a function which has complex CFG, the common solution is to only emulate the most related block(s), not the whole function. Sometimes the emulation might be done multiple times for separate code snippets. In summary, the design philosophy of LWE-based IoC extraction is simplicity and flexibility. Equipped with proper behaviour patterns, it can easily be applied to other botnet families.

4. MIRAI CODE IN GAFGYT

Since both target the same set of *Linux* devices and both have had their source leaked, the code sharing between Mirai and Gafgyt is very common, especially in infection vectors and integrated exploits [3, 4]. While it might be difficult to figure out every piece of shared code from Mirai in a Gafgyt sample, in most cases it's possible to check the widely used Mirai code in a Gafgyt sample with Mirai's characteristic encrypted configurations that are tightly connected with features including scanning, killer, random string generation, and attacking [5]. When other botnet authors decide to borrow code from Mirai, the configuration-related data and code usually have to be copied together because it's difficult to separate them from the really wanted features. Those traces make it possible to research how Mirai code is used in Gafgyt by studying the extracted configurations.

In summary, Mirai configurations have been successfully extracted from 3,700 Gafgyt samples with the automatic configuration extraction scheme introduced in [6]. They are grouped in the combination of item count, total size, and branch name, with the stats illustrated in Table 2.

In Mirai, each configuration is numbered and is used in a manner of 'unlock-retrieve-relock', as shown in Figure 8. Therefore, the heavily used items can be figured out by tracking the related functions, e.g. table_retrieve_val.

```
// Print out system exec
table_unlock_val(TABLE_EXEC_SUCCESS);
tbl_exec_succ = table_retrieve_val(TABLE_EXEC_SUCCESS, &tbl_exec_succ_len);
write(STDOUT, tbl_exec_succ, tbl_exec_succ_len);
write(STDOUT, "\n", 1);
table_lock_val(TABLE_EXEC_SUCCESS);
```

Figure 8: Mirai 'unlock-retrieve-relock' style of configuration referencing.

Items_size_branch	samples
36_412_KYTON	3,346
39_437	188
39_417	63
36_431_KYTON	36
36_428_Reaper	14
23_312_REKAI	14
4_80	6
23_283_REKAI	5
24_394_REKAI	3
15_201_ROOT	3
24_302	2
10_118	2
34_384_KYTON	1
22_296_REKAI	1

Table 2: Stats of Mirai configurations in 3,700 Gafgyt samples.

Popular groups of configurations in Table 2 have been manually investigated. The key findings are summarized as follows:

1. Each group of samples shares the same configuration usage patterns. In most cases they can be classified as the same variant.
2. It's strange that in the largest group of 36_412_KYTON, only one configuration item is used for random string generation, which has an index of 0x26 in Figure 9.
3. The groups of 39_437, 39_417, 36_431_KYTON and 36_428_Reaper are very similar in both content and configuration referencing patterns. The configurations are widely used in scanner, killer, and random string generation. The four groups of samples are probably derived from the same code branch.

```
[0x01]: "C\", addr=0x00000001, size=2
[0x02]: "(null)\x00", addr=0x00000002, size=7
[0x03]: "/dev/watchdog\x00", addr=0x00000003, size=14
[0x04]: "/dev/misc/watchdog\x00", addr=0x00000004, size=19
[0x05]: "/dev/watchdog0\x00", addr=0x00000005, size=15
[0x06]: "/bin/watchdog\x00", addr=0x00000006, size=14
[0x07]: "/etc/watchdog\x00", addr=0x00000007, size=14
[0x0a]: "shell\x00", addr=0x0000000a, size=6
[0x0b]: "enable\x00", addr=0x0000000b, size=7
[0x0c]: "system\x00", addr=0x0000000c, size=7
[0x0d]: "linuxshell\x00", addr=0x0000000d, size=11
[0x0e]: "\xe2\xe1\xe8\x80", addr=0x0000000e, size=4
[0x0f]: "sh\x00", addr=0x0000000f, size=3
[0x10]: "ncorrect\x00", addr=0x00000010, size=9
[0x11]: "ogin\x00", addr=0x00000011, size=5
[0x12]: "enter\x00", addr=0x00000012, size=6
[0x13]: "assword\x00", addr=0x00000013, size=8
[0x14]: "/bin/busybox KYTON\x00", addr=0x00000014, size=19
[0x15]: "KYTON: applet not found\x00", addr=0x00000015, size=24
[0x16]: "/proc/\x00", addr=0x00000016, size=7
[0x17]: "/exe\x00", addr=0x00000017, size=5
[0x18]: "/fd\x00", addr=0x00000018, size=4
[0x19]: "/maps\x00", addr=0x00000019, size=6
[0x1a]: "/proc/net/tcp\x00", addr=0x0000001a, size=14
[0x1b]: "0\x16\x00\x17H$\x02\x00", addr=0x0000001b, size=8
[0x1c]: "/dev/null\x00", addr=0x0000001c, size=10
[0x1d]: "STD\x00", addr=0x0000001d, size=4
[0x1e]: "/proc/net/route\x00", addr=0x0000001e, size=16
[0x1f]: "/proc/net/tcp\x00", addr=0x0000001f, size=14
[0x20]: "/proc/self/exe\x00", addr=0x00000020, size=15
[0x21]: "UPX!\x00", addr=0x00000021, size=5
[0x22]: "/proc/net/route\x00", addr=0x00000022, size=16
[0x23]: "/etc/rc.d/rc.local\x00", addr=0x00000023, size=19
[0x24]: "/bin/sh\x00", addr=0x00000024, size=8
[0x25]: "-\x0a\x02\x01\x07\x10\x01\x00", addr=0x00000025, size=8
[0x26]: "qC8cVuGTnRH6cfv7sjcYPFv7guAmZxbQRc57fV77IUUj5b6wocpfFJPmHC\x00", addr=0x00000026, size=59
```

Figure 9: Extracted configurations for 36_412_KYTON.

5. CONCLUSION

I have introduced how to extract the IoCs of Gafgyt's register message and C2 by detecting its characteristic C2 loop and lightweight emulating the initConnection function. The C2 loop and initConnection function can also be used for variant tracking and classification. Meanwhile, using the techniques introduced in Section 2 and 3, new samples beyond the types illustrated in Appendix B and C can also be analysed.

I also introduced the general ideas of LWE-based data extraction. Compared with sandbox-based dynamic analysis, a LWE-based solution is easier to deploy and more flexible. Equipped with new behaviour patterns, LWE-based data extraction can also be applied to other botnet families.

Finally, I investigated the widely used Mirai code in Gafgyt by extracting Mirai characteristic encrypted configurations. It shows Mirai code can be tracked by analysing its configuration. The tracked code also helps to classify Gafgyt variants.

REFERENCES

[1] BASHLITE, https://malware.wikia.org/wiki/BASHLITE.

[2] libemu – x86 Shellcode Emulation, http://libemu.carnivore.it/.

[3] Tweets by @0xrb, https://twitter.com/0xrb.

[4] Tweets by @bad_packets, https://twitter.com/bad_packets.

[5] Mirai source code. https://github.com/jgamblin/Mirai-Source-Code/tree/master/mirai.

[6] Liu, Y.; Wang, H. Tracking Mirai Variants. Proceedings of the Virus Bulletin International Conference 2018. https://www.virusbulletin.com/virusbulletin/2018/12/vb2018-paper-tracking-mirai-variants/.

APPENDIX A: SHA256 OF SAMPLES

MD5 hash: 00b310f837972e972d12dea0661302f3

SHA-256 hash: 14b626834274d346f67e04849a5409c8710bfccc2cb718dbcc4995fab5e451fb

MD5 hash: 02cc10ebf07c6f70b3437340bec1a265

SHA-256 hash: 372a4b0f5a347fb8d6642c88aa89793fd5efe71d577db619d8a7ddab18133311

MD5 hash: 08e57e7ed679df8cd9891f596ba8d8ca

SHA-256 hash: 4e5759b33d3be016bf6f58cb080539f83085a7c51c451d15bce9aadf99773cb2

MD5 hash: 0967a1ad0056ca664e064a59e9f263e1

SHA-256 hash: 68c20f82fe5385458a9fc6539b2dd5928ea34d039c182218292d224e61be8d1a

MD5 hash: 0000e22a5cd366b112a0f1112c565ac7

SHA-256 hash: b5b7effe9052e9e1669ed6a14c72ecd20080ee0d57d8d3e4759061d75f7e5c09

MD5 hash: 00037f246c41482b7175201c515e2a1c

SHA-256 hash: a2d787f4d0d46a88778d31498c5f2ce49e981fa6201ebc4223d7079c7bb86e7c

MD5 hash: 0003d90a31eb72caf045ea7f622d4dc5

SHA-256 hash: d4238cf7504d41bd11ebbbaec60bea6a2b9d8d136325bfbb498df4f0b3ab215f

MD5 hash: 000a6673dcde7dfd646fdc946a3e305f

SHA-256 hash: 92abff4b88db8fcdb9936f29a101c800b3d3402b7ce6c313a2149e23df76dc6f

MD5 hash: 001618368ffd8735837267d9763b0fa1

SHA-256 hash: 51caa96d031214644e1ebf7604983f4da4eb95c95a4efd4f8967319780bb1fa3

MD5 hash: 00432f33fb3f5cc5377266a5439567bf

SHA-256 hash: a30c66532a7e54fa8484c0ef36d93a6ffa14cb923981254846e73db9a444e95f

MD5 hash: 005cec5d3928f5c5534f4b46989029a8

SHA-256 hash: 811e56ef2dc70b5b44c94b292248ee25cf44a51f99751755febd2bb34d92a3aa

MD5 hash: 05536b105070a1aeb6ceeee4dd5043c8

SHA-256 hash: 9ef45c92c861014ff3011e2ad7b774c62a878893530b6da5603cf0b79517fe9e

MD5 hash: 056470bb9476d108d8042a5b5c70d1b9
SHA-256 hash: 5dfa3c6d09679fde54184c34d8f0d11fbafe5bb3d4b301cded60cf7b55a61d1b

MD5 hash: 09501d91c5adccc72c1884a5d931eb9c
SHA-256 hash: 4e5e86ed730ff2ced1753798f9f184e316f831dd75f64001653e1b4ac00b7763

MD5 hash: 0a115b05c9508672068c82c52e538028
SHA-256 hash: a6051ce614dabba21e510a8013dc8580d427c86ca50a9b73501e3fe0b524dc04

MD5 hash: 142c0e7d864fa156b4622062cca27f8d
SHA-256 hash: 608a455651e6fbba5caeedf71e29e4731f01d38e20dd0acd074052bfc0e00d53

MD5 hash: 23c0f0ea828c65b3e2502c9cfc24f91f
SHA-256 hash: 77193fb6262e7f8908f45aba9b1c9bc3aa7d997830aef6254665457430d72eb4

MD5 hash: 3cda17370a9c72120001c44fba76b442
SHA-256 hash: a3b38780263a927af94692c752d834df6b739e1621f1652e677d6541a0529c14

MD5 hash: 44e8a1908f234eb43ef081652d17e8a0
SHA-256 hash: fb64c45838237be9ff218a0d96e9f16985013e75e72b9802d3428084e53176af

MD5 hash: d5a54f8d85dd2e653ef4ce1533e0cc9c
SHA-256 hash: 41c8961f18e54973a11e3ae34b4bbd8a889a69655eaa0d6907adb63e11c0466e

MD5 hash: 5264b21d93ce4668c3f8aac823924c04
SHA-256 hash: 55e2e2c8f4c24c83f3f9d7eda7b3a3c4c85879bc4326cc27e901bc651dd518db

APPENDIX B: SIX TYPES OF C2 LOOPS

Type 1

Criteria:

1. There are four blocks included in the loop.
2. Only one function is called in the first block.
3. At least two functions are called in the second block.
4. At least one string is referenced in the second block.
5. Only one function is called in the third block.

Example 1: MD5=0000e22a5cd366b112a0f1112c565ac7, mips32

```
"[initConnection][] -> [getBuild, sockprintf]["mips", "arch %s"] -> [recvLine][] -> [][]"
```

Example 2: MD5=000a6673dcde7dfd646fdc946a3e305f, x86

```
"[fcn.0804d08d][] -> [fcn.0804d3cb, fcn.0804882a]["BUILD %s"] -> [fcn.0804916b][] -> [][]"
```

Type 2

Criteria:

1. There are four blocks included in the loop.
2. Only one function is called in the first block, and it must match the initConnection criterion.
3. No functions and strings are referenced in the second block.
4. Only one function is called in the third block.

Example 1: MD5=0003d90a31eb72caf045ea7f622d4dc5, arm32

```
"[initConnection][] -> [][] -> [recvLine][] -> [puts]["LINK CLOSED"]"
```

Example 2: MD5= 0967a1ad0056ca664e064a59e9f263e1, x86

```
"[fcn.0804d14f][] -> [][] -> [fcn.08048a8e][] -> [][]"
```

Type 3

Criteria:

1. There are three blocks.
2. Only one functions is called per block.
3. The function called in the second block must match the initConnection criterion.

Example 1: MD5=00037f246c41482b7175201c515e2a1c, arm32

```
"[fork][] -> [initConnection][] -> [sleep][]"
```

Example 2: MD5= 02cc10ebf07c6f70b3437340bec1a265, x86

```
"[fcn.0804db3c][] -> [fcn.0804d16b][] -> [fcn.08050b60][]"
```

Type 4

Criteria:

1. There is a string of 'fftt:%s' referenced in the loop.
2. There must be functions called in the first or second block, and there must be one function matching the initConnection criterion.

Example 1: MD5=005cec5d3928f5c5534f4b46989029a8, arm32

```
"[initConnection][] -> [sprintf, sockprintf]["fftt:%s"] -> [recvLine][] -> [][]"
```

Example 2: MD5= 00b310f837972e972d12dea0661302f3, arm32

```
"[][] -> [fcn.00009a8c][] -> []["fftt:"] -> [fcn.0000c5c0, fcn.0000c4e0]["idk"] -> [fcn.00009f24]
[] -> [fcn.0000ae70][]"
```

Example 3: MD5=0a115b05c9508672068c82c52e538028, x64

```
"[fcn.004003f1][] -> [fcn.00403420, fcn.00400a3a, fcn.00401c0b]["fftt:%s"] -> [fcn.004030c8][] ->
[][]"
```

Type 5

Criteria:

1. There is only one block.
2. At least one function is called in the first block, and one of them must match the initConnection criterion.

Example 1: MD5=23c0f0ea828c65b3e2502c9cfc24f91f, x64

```
"[Connection,recv_buf][]"
```

Example 2: MD5= 08e57e7ed679df8cd9891f596ba8d8ca, arm32

```
"[Connection, botkiller, recv_buf][]"
```

Example 3: MD5=142c0e7d864fa156b4622062cca27f8d, x64

```
"[fcn.00404420][]"
```

Type 6

Criteria:

1. It does not match types 1 to 5.
2. At least one string containing '%s' is referenced in the loop.
3. At least one of the called functions matches initConnection.

Example 1: MD5=056470bb9476d108d8042a5b5c70d1b9, arm32

```
"[initConnection][] -> [][] -> [__GI_strchr][] -> [getDistro, access]["/etc/ssh/"] -> [access]
["/etc/dropbear/"] -> [__GI_asprintf]["ARM4", "AUTH %s %s %s.%s %d %s %s %s", "1.7.9"] ->
[sockprintf, free][] -> [recvLine][]"
```

Example 2: MD5=05536b105070a1aeb6ceeee4dd5043c8, x86

```
"[initConnection][] -> [getBuild, sockprintf]["unknown", "arch %s"] -> [sockprintf]["unknown",
"Multiple Processors Detected: Starting Scanners %s"] -> [][] -> [recvLine][] -> [][]"
```

Example 3: MD5=09501d91c5adccc72c1884a5d931eb9c, x86

```
"[fcn.08048f40][] -> [fcn.0804e0bf]["22", "/usr/bin/python"] -> [fcn.0804e0bf]["/usr/bin/
python3"] -> [fcn.0804e0bf]["/usr/bin/perl"] -> [fcn.0804e0bf]["/usr/sbin/telnetd"] -> []
["Unknown Port"] -> [fcn.0804f5a8, fcn.08048d40]["x86_32", "[0m[[36mKanashi[0m][[36m%s[0m]
[36m:[0m[[36m%s[0m] [36m>[0m [[36m%s[0m]"] -> [fcn.08048750][]"
```

APPENDIX C: SIX TYPES OF INITCONNECTION

Type 1

MD5= 00432f33fb3f5cc5377266a5439567bf, x86

Simplified pattern: 'cw4cccw4c'

Behaviour pattern: 'call_memset, w4, call_strcpy, call_strchr, call_socket, w4, call_connectTimeout'

Static pattern: blocs=11, edges=14, called_functions=7, strs=["198.134.120.150:23"]

Extraction rules:

Reading global memory pointed by arg2 of strcpy() to get the string format of 'C2:port'

Type 2

MD5= 001618368ffd8735837267d9763b0fa1, x64

Simplified pattern: 'cw4cccw4c'

Behaviour pattern: 'call_memset, w4, call_sprintf, call_strchr, call_socket, w4, call_connectTimeout'

Static pattern: blocs=11, edges=14, called_functions =7, strs=["%d.%d.%d.%d"]

Extraction rules:

1. IP is generated by concatenating arg3 to arg6 of sprintf with the format of '%d.%d.%d.%d'.
2. Port is read from arg3 of connectTimeout.

Type 3

MD5= 3cda17370a9c72120001c44fba76b442, arm32

Simplified pattern: 'cccw4cccc'

Behaviour pattern: 'call_fcntl, call_fcntl, call_socket, w4, call_inet_addr, call_connect'

Static pattern: blocs=10, edges=14, called_functions =7, strs=["167.71.33.152"]

Extraction rules:

1. C2 IP is read from the global memory pointed to by arg of inet_addr.
2. Port is read from the memory pointed to by arg2 of connect() according to the definition of struct sockaddr.

Type 4

MD5= 44e8a1908f234eb43ef081652d17e8a0, x64

Simplified pattern: 'cccw4cccc'

Behaviour pattern: 'call_fcntl, call_fcntl, call_socket, w4, call_htons, call_htonl, call_connect, call_puts'

Static pattern: blocs=4, edges=4, called_functions =7, strs=["Unable To Connect! ", "Succesfully Connected! "]

Extraction rules:

1. C2 IP is read from arg1 of htonl(), while port from arg1 of htons().

Type 5

MD5= d5a54f8d85dd2e653ef4ce1533e0cc9c, x86

Simplified pattern: 'cccw4cccc'

Behaviour pattern: 'call_fcntl, call_fcntl, call_socket, w4, call_htons, call_inet_addr, call_connect, call_puts'

Static pattern: blocs=4, edges=4, called_functions =7, strs=["Unable To Connect! ", "Succesfully Connected! "]

Extraction rules:

1. C2 IP is read from arg1 of inet_addr(), while port from arg1 of htons.

Type 6

MD5= 3cda17370a9c72120001c44fba76b442, arm32

Simplified pattern: 'cccw4ccc'

Behaviour pattern: 'call_fcntl, call_fcntl, call_socket, w4, call_inet_addr, call_connect'

Static pattern: blocs=10, edges=14, called_functions =7, strs=["167.71.33.152"]

Extraction rules:

1. C2 IP is read from the global memory pointed to by arg of inet_addr.
2. Port is read from the memory pointed to by arg2 of connect() according to the definition of struct sockaddr.