HIDE’N’SEEK: AN ADAPTIVE PEER-TO-PEER IOT BOTNET

Adrian Şendroiu & Vladimir Diaconescu
Bitdefender, Romania

{asendroiu, vdiaconescu}@bitdefender.com

ABSTRACT

Along with the rise of Internet of Things (IoT) products and technologies comes the growth and evolution of IoT botnets; the impact of Bashlite, Mirai and Reaper, to name a few, are a testament to that fact. This paper presents a thorough analysis of the inner workings of Hide’n’Seek (or HNS), a peer-to-peer botnet discovered in January 2018. With an exploit table that can be updated in memory and modular in its approach, HNS gives us a glimpse of what kinds of IoT threats we will encounter in the years to come. Starting from a humble list of 12 infected machines, it has undergone a few updates and reached tens of thousands of victims around the world. While this particular botnet has amassed an impressive number of compromised devices, its more interesting characteristics lie with other novelties and peculiarities discovered during our investigation.

In contrast with other botnets, which rely on a centralized, asymmetric architecture with one or more C&Cs and multiple bots, HNS uses a custom-built peer-to-peer system in which any peer can both issue and receive commands. This somewhat different approach to the traditional IoT botnet landscape brings about new challenges moving forward. For instance, some of the design choices, such as the P2P model, lead to an increased difficulty in analysing and taking down such a threat. One notable feature is the presence of a dynamic table of exploits as well as a reputation system – knowledge – among peers, which allows for new exploits to be added and spread autonomously through the network.

Even though the capabilities of the botnet, such as propagation (worm-like behaviour), peer-discovery, data exfiltration and modularity, are laid bare, the intent, origin and business model of the botnet are subject to speculation, since it oddly features no DDoS elements at the time of our investigation. However, such elements may easily become available in potential updates to the expanding botnet.

1. INTRODUCTION

Over the past couple of years, IoT botnets have amassed staggering numbers of compromised devices and broken records in terms of DDoS capabilities. While each of these botnets features its own peculiarities, the architecture tends to follow the same preset:

- One or more mirrored C&Cs
- An IP scanner
- A ‘bot killer’ to find and eliminate competing bots running on the victim device
- A dictionary of default credentials
- Sometimes one or two CVEs
- A list of commands and attack types

The main goal of these botnets is performing DDoS attacks for profit. In some cases, the malicious code running on the victim can execute arbitrary system commands or relay information about the victim device to the C&C. These types of botnets have been successful over the years, despite the fact that most of them share the same code and functionality. However, the IoT threat landscape is starting to diversify in both scope and approach.

In this paper, we present Hide’n’Seek, a new kind of botnet, relying on a custom-built P2P protocol to spread commands and updates through the network of victims, doing away with C&Cs altogether. We found this botnet in our Telnet honeypots and took interest in it, as it stood out in its lack of similarity with any of our samples.

2. ANALYSIS

In this section, we will provide an overview of Hide’n’Seek, from both historical and functional points of view.

We first saw this bot in early January 2018, having successfully infected one of our honeypot systems. Due to it presenting almost no similarities to any other botnet, this new sample immediately caught our attention. The following are the usual questions that come to mind when encountering a new botnet:

- What is its C&C?
- What capabilities does it have?
- What kinds of systems is it targeting?
- Are there any IOCs?
- Who is the author?
- How can it easily and automatically be identified?
- How do we keep an eye on it?

In order to answer these questions, we thoroughly reverse engineered the sample and found 14 IP addresses hard coded in the binary, 12 of which pointed to South Korea. 14 C&Cs is not unheard of, but it was atypical nonetheless. As we delved deeper into our analysis, it slowly became apparent that this new botnet operated under a different ruleset than we were used to.

2.1 Infection process

First and foremost, from a victim’s perspective, how does an IoT device become infected with this new strain of malware? If the victim device has port 23, 2323, 9527, 80 or 8080 open, then at some point in time it will receive a connection from an infected machine. If the attacker finds any one of the aforementioned ports open, a dictionary attack will be attempted using default credentials found on common IoT devices over Telnet.

If the attack manages to gain access, then it moves on to the next phase, in which it tries to determine the type of system it managed to break into. This is achieved by issuing system-specific commands which uniquely identify the victim’s device type.

The final phase consists of the attacker dropping and running the malicious binary, compiled specifically for the victim’s architecture, through various attempts which will be covered in a subsequent subsection.

2.2 A different architecture

The binary is programmed in C and compiled for all major platforms as well as for SPARC, Motorola 68000 and SuperH. Apart from a few functions which are common across almost all IoT botnets, most of the functions are unique to this particular botnet.

During our analysis, we noticed a few telltale signs which pointed us towards the notion that this was, in fact, a peer-to-peer botnet:

- The 14 hard-coded C&Cs all had random five-digit ports. When mirroring C&Cs, it is common to keep the port fixed.
- All the commands which were received from the C&Cs were also issued by the binary itself. There were no one-directional commands present.
- After a thorough investigation, the C&Cs turned out to be IoT devices, not VPSs, identical in capabilities with the victim.
- The binary would update the list of C&Cs at runtime.

2.3 Life as a peer – P2P protocol and capabilities

From a victim's perspective, what does being part of this botnet entail? Firstly, the dropped binary receives a series of command-line arguments, as listed below, through which its initial configuration is extended with further options.

The newly awakened bot will first gather information about the system on which it is running, such as available memory, files and local interface IPs. The former is used to determine how much memory to allocate for its dynamic list of peers.

Option name	Argument	Description
k	Port number	Kills all processes listening on the specified port
l	Port number	Listens on this port for P2P communication
s	Path	Adds the given path to the hashtable
a	[IP:PORT]	Supplements the list of starting known peers
e	IP:PORT	Manually provides a target to be scanned

Once the set-up is complete, the binary listens on a random UDP port, or the one specified through the command-line, and then starts its components. The functionality is roughly split into two main subroutines: one that scans the Internet for more vulnerable devices and one that handles the peer-to-peer communication.

There are two data structures related to the peer-to-peer protocol: a config cache and a data cache. The config cache is a mapping between a payload ID and a SHA512 hash, while the data cache maps between a SHA512 hash and some arbitrary data. Some data cache entries can have an additional flag which indicates that the entry is a file path.

In this context, a payload ID is a number that identifies the architecture of a particular payload. For example, 0x15000000 denotes a payload for the ARM architecture, 0x14000000 is MIPSLE, 0x13000000 is MIPSBE, and so on.

The config cache is initially empty, while the data cache is initialized with a single entry representing the contents of the executable file itself. Both these caches are further populated via the P2P network.

The communication protocol is implemented over UDP and consists of nine types of messages, where the first character of the message falls into the ASCII range and identifies the message type. We can group these messages into several types, according to their functionality.

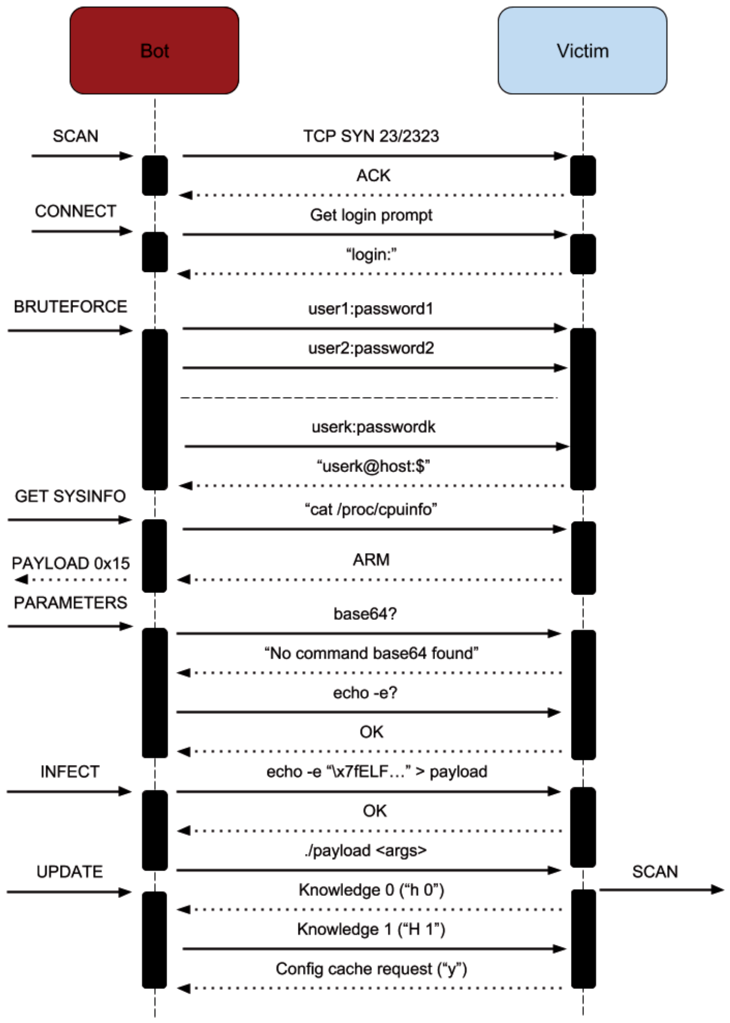

Figure 1: HNS infection.

1. *Data exfiltration: 'm', 'y' and 'Y':* These commands allow data transfers between peers. A peer can query another peer for the contents of the config cache, data cache, or an arbitrary file on disk. Data is usually transferred in 256-byte chunks.

   'm' and 'y' are used to initiate a data transfer. They have the following format:

   ```
   'm' + u8[64](hash) + u32(ip) + u16(port) +
   u16(id) + u8(hops)
   ```

   ```
   'y' + u8[64](hash) + u16(chunk_index) + u16(id)
   ```

   Both commands include the hash of the data being requested. The 'm' command initiates a transfer, while 'y' is used to request subsequent chunks (although 'y' can be used to initiate a transfer as well, by requesting the chunk with the index 0). Another difference is that an 'm' message can be further broadcast if the hash is not found locally (the IP:PORT and hops parameters are used in this regard). If the hash is an array of zeros then the contents of the config cache are sent out. Otherwise, the hash is searched for in the data cache. If the entry is a file path, then the contents of that file are sent out, otherwise the bot will just send the entry value.

   'Y' is the reply message, sent as a response to both 'm' and 'y':

   ```
    'Y' + u16(chunk_index) + u16(id) + u8[](data)
   ```

   The ID is a message identifier field which is copied verbatim from the ID in the request messages.

   Once a transfer has been completed, there are multiple possibilities:

   - If the data is a new config cache, it is verified (via ECDSA – a public key is hard coded inside the binary). If the verification passes, it is installed as the new config cache.
   - If the data is a new binary, then it is dropped on the disk and executed – this serves as an update mechanism.
   - Otherwise, the data will just be inserted into the data cache.

2. *Peer discovery: '~' and '^':* These are used to keep the peer list up to date: when a peer receives '~', it will pick a random IP:PORT pair from its peer list and send them back in a '^' reply. Upon receiving the '^', the requester will insert this new peer into its table. All peers periodically send '~' messages to each other.

3. *Payload discovery: 'h' and 'H':* The config cache is an important data structure, since the bot will query it for a payload whenever it needs to infect another target. For this reason, the bot implements a mechanism which ensures that more recent versions of the config cache are propagated to every participant in the network. To achieve this, the config cache is tagged with a version number (knowledge number).

   The higher this number, the more recent the config. Each bot tries to obtain a newer config via the 'h' and 'H' messages:

   - 'h' + u32(version): If the received knowledge number is larger than the current one, the bot will request the config cache (via 'y') from the sender of this message. Otherwise, it will reply with an 'H' + its own version.
   - 'H' + u32(version): If the received knowledge number is larger than the current one, the bot will request the config cache (via 'y') from the sender of this message. This is quite similar to 'h', except that it does nothing when the current config version is greater than the received one.

   The bot will periodically send 'h' messages to other peers in the network. In particular, this is how a newly infected machine obtains a config: since the config cache is initially empty, the machine will keep sending 'h' + '\x00\x00\x00\x00' messages throughout the network. Eventually, a peer with a larger config cache will reply with an 'H', causing the newly infected peer to request the config from it.

4. *Other: 'z' and 'O':* When a vulnerable device is found by the scanner, its IP address and listening port are sent to a random peer via a 'z' message. 'O' is an acknowledgment message, sent as a reply to 'z' and a few other commands as well.

Command prefix	Argument(s)	Description
h	Knowledge value	Knowledge query
H	Knowledge value	Knowledge reply
m	hash, IP, PORT, seq, hops, ID	Data request
y	hash, chunk index, seq	Data chunk request
Y	chunk index, seq, data	Data chunk reply
~	none	Peer request
^	flags, IP, PORT	Peer response
O	checksum	ACK
z	IP, PORT	Report vulnerable device

2.4 Scanning for new victims

Apart from communicating with the rest of the network, receiving peer, exploit and binary updates and discarding non-responsive peers, each infected machine scans the Internet for vulnerable devices to compromise. The scanner structure is similar to that of those found in other botnets, such as Mirai: in order to achieve greater scanning speeds, the bot first sends a bunch of TCP SYN packets to random targets, using a non-blocking raw socket, on port 23, 2323, 9527, 80 or 8080 (one of these values picked at random, with a bias towards 23). For each host that answered the initial SYN, the bot will open a new connection using a regular socket and continue the communication.

Subsequent communication is also done asynchronously, using non-blocking sockets. For each host, the bot will maintain a connection structure and will use a state machine to navigate among the possible requests/replies. Although the state machine is different depending on whether the scanning port was 23/2323/9527 (Telnet) or 80/8080 (HTTP), the flow is quite similar: first try to gain access on the remote system, then try to deliver an appropriate payload. In the case of Telnet, the bot will first look at the server prompt. In particular, there is a special case when the prompt hash matches some hard-coded values, in which case a specific set of credentials is attempted. Otherwise, a pair of credentials picked randomly from a hard-coded dictionary is used. Since most default credentials are in the form of root:password or admin:password, HNS uses a clever scheme for storing them: it keeps only the password with either the second or the third byte altered (MSB set to 1, illegal for an ASCII character). If the second is altered, then root is used as the user, otherwise admin is used. Of course, the altered byte is restored (MSB cleared) before using the string. If no bytes are altered, then the credential has a different user, which is stored in the dictionary.

Once access has been gained, the bot will attempt to escalate privileges if by any chance the system is vulnerable to CVE-2016-10401, by issuing a 'su' command with the password 'zyad5001'. Then it will proceed to determine the system architecture by checking the ELF header of /bin/echo and by looking in /proc/cpuinfo. The purpose of this step is to obtain a payload ID, as described in the P2P protocol section. For example, if the system is detected to be an ARM one, payloadID 0x15000000 is selected.

The next step is to find some suitable payload delivery method. This is done by probing the existence of base64 or by checking whether echo -e works. A separate case is when both the bot and its victim are in the same LAN, in which case, besides the methods previously mentioned, a delivery via tftp or wget is also attempted (the bot implements a minimal HTTP and TFTP server for this purpose).

Now the bot will proceed to delivering the payload. Starting from the selected exploit ID, it will query the config cache to obtain the SHA512 hash of the payload, and then query the data cache to obtain the actual payload data to be delivered. It is important to note that a freshly started bot will fail at this step, and will not infect the target system, even though it was able to obtain access. This is because its config cache will be empty, and so it will be unaware of what payload to deliver. Thus, a bot that has not had the chance to obtain a config cache from another peer will not be 'infectious'.

The last step is to run the payload. One notable feature here is that the bot will make use of the 'a' command-line argument, described in the previous section: for each active peer in its list, an 'a' argument with that particular IP address and port will be appended to the command line. This is to ensure that the freshly started executable on the victim machine has a set of active peers from which to bootstrap its config and data caches.

In the case of HTTP, the probing is done by issuing a GET request. Then, the HTTP answer will be scanned for strings such as ': Basic realm="NETGEAR"' or ': Basic realm="TP-LINK". For *Netgear* routers, the bot will try to gain remote code execution by running the setup.cgi exploit, while for *TP-LINK*, the '/userRpmNatDebugRpm26525557/start art.html' exploit will be attempted. The minimal HTTP and TFTP server implementations will also come in handy here, since the commands that the bot will inject will rely on either wget or tftp.

2.5 Updates

HNS underwent three updates over the course of five months; the first update removed the initial list of peers from the binary. This time, the initial starting list used only the peers supplied by the infecting machine when running the binary on the victim. This update also changed some of the bytes used in the protocol, as well as some of the internal structures used by the bot. Despite these updates, the new sample did not manage to get off the ground, since it suffered from the same drawbacks as the previous one.

The second update changed the hashing function to SHA512, changed the ECDSA key, and also introduced a periodic check to see whether neighbours are still active, cleaning them up otherwise. It also re-added the hard-coded list of IPs as a fallback mechanism; if a peer becomes inactive, then it is replaced with one of these more reliable peers.

In May 2018 we noticed another update. Some parts of the code were refactored, but some new functionality was also added:

- The bot now features a persistence mechanism: before deleting itself, the binary is copied to /etc/init.d or /etc/rc.d.
- The list of HTTP exploits was extended with some exploits for *Linksys* routers and some IP cameras from *Avtech* and *Wansview*.
- The bot is now capable of dropping and executing other binaries. We have observed a particular case in which a Monero miner was dropped.
- Besides the previously mentioned payload delivery methods, the capability to deliver a uuencoded payload was added, provided that the target system has the uudecode command-line tool.

2.6 Who is the author?

We investigated 12 of the 14 IP addresses hard coded in the first iteration of HNS, which we found to belong what we believed to be a dozen IP cameras in South Korea.

The peer-to-peer aspect of the botnet makes it difficult to pinpoint the people behind it; instead of a singular C&C which uses a clear to determine hosting service, with an owner name attached to it, we have tens of thousands of similar devices. In order to get closer to uncovering the origin of the botnet, one would have to voluntarily keep a number of devices available for infection and carefully instrument the binary in order to catch any changes in process memory. Even more problematic is the fact that these infiltrated devices would have to reach a high neighbour count within the network, or to be part of multiple disjoint groups of the network, in order to have a chance for file exfiltration requests to reach them.

3. MONITORING

By allowing some of our honeypots to become infected with HNS, and by keeping them running and dumping memory contents, we managed to gather a starting list of likely active peers. We then sent peer requests to these and added newfound peers to the pool as they were being discovered. Since the most prevalent botnets at the time were families such as Mirai, and HNS was still in its initial stage, our honeypots were constantly being infected with Mirai and other types of more ‘popular’ bots. To overcome this challenge, we fine-tuned our systems to wait specifically for HNS and block out anything else.

Due to a few significant flaws in the design of the P2P network, the first iteration quickly came to a halt under its own weight. The protocol did not periodically check whether or not the peers were still active, therefore peers could not clean up their lists of neighbours.

A peer was identified via IP address and port. While this is good (from the P2P network’s perspective) in cases in which multiple victims are behind NAT, this can end in the following scenario:

- A new peer comes online on IP1:PORT1 and is added to the list of neighbouring peers, then relayed throughout the network.
- The owner of the infected device resets it, or there is a power outage in the area.
- Having no persistence mechanism, the device will be clean for some time before it becomes infected again, and comes online as a new peer with IP1:PORT2.
- At this point, the network has added redundant information which it cannot check or get rid of.
- As we have shown in our analysis section, if a newly spawned bot cannot contact its peers, then it cannot infect new victims, and the entire process draws to a halt.

Such scenarios lead to the propagation of dead peers throughout the network.

3.1 Determining the size of the botnet

There were a few challenges which we needed to overcome in order to accurately track and measure the botnet. When issuing a peer request to a bot, it gives back a peer’s IP address and port from its list, chosen randomly based on the current system time and the soliciting IP:

```
idx = (rand() + IP) mod NUMPEERS;
peer = peerlist[idx];
```

It follows that if it receives two requests from the same IP address in less than a second, then it will send back the same peer. This means that requests to a single peer need to be spaced out by one second in order to efficiently collect its list of neighbours.

Because the maximum number of peers stored in memory is determined dynamically, the only way to determine the number of peers a particular bot knows about is to query it until it starts repeating itself in a consistent manner. This also has the side-effect of learning a machine’s available memory prior to infection through the number of stored peers. Lastly, whenever an unknown IP address sends a peer request to a bot, the bot adds the unknown IP to its list of peers. This leads to two problems:

- The queried bot will attempt to further exchange information with this new peer, such as reputation value and neighbour information. To overcome this, one must close the socket used and create a new one, listening on a different port, when changing peers. Otherwise, one risks being flooded with traffic from, eventually, the entire botnet.
- If you query the entire botnet from peer to peer, it follows that your IP address becomes the most common address in the entire network. We tackled these challenges by carefully spacing out our queries and prioritizing newly discovered peers, as well as the least recently queried peers.

4. PURPOSE

One peculiar aspect of HNS is that it features no DDoS capabilities. However, there are many other goals that this botnet has set out to achieve. We can build a picture of the author’s business model based on what we have detailed so far.

4.1 Data exfiltration

As seen in our analysis, HNS is capable of transferring files through the network. The list of target devices includes IP cameras and routers. The botnet could easily be used for mass surveillance, theft of private keys, or to take advantage of any other leverage provided by having access to the filesystems of these devices.

4.2 Botnet as a service

Peers can also receive and run arbitrary binaries. This feature suggests the idea that the author could easily rent the botnet in its entirety to someone interested in running a DDoS attack, but without the hassle and risks involved in growing their own botnet from scratch.

This possible direction has formed a more solid foundation in the update of May 2018, when HNS started running a Monero miner on its victims. It is unclear whether the wallet for the mined coins belongs to the authors themselves or a different party.

4.3 Future possible functionality

The P2P design of HNS offers great flexibility. The author could easily issue an update that would provide the binary with DDoS capabilities. Traffic could also be proxied and chained through any number of peers in the botnet, making the act of determining the point of origin of an attack very difficult.

CONCLUSION

The analysis of Hide’n’Seek has taken us in a new direction for IoT botnets. By incorporating multiple infection vectors, it has managed to reach nearly 100k connected victims without the use of C&Cs and unattended. We have presented this new botnet’s history and life cycle, which brings to light a new set of security concerns and challenges for the future. This new threat can exfiltrate files throughout the network, extend the list of exploits it uses to find new victims, or even replace itself with an entirely different malicious binary, changing the behaviour of the entire botnet. We hope that this silent and seemingly harmless botnet will raise awareness for organizations and individuals alike and motivate them to search for and find adequate measures to counter threats such as this and the ones that will surely follow.