(U) Hive Engineering Development Guide

September 24, 2013

Classified By: 0706993
Reason: 1.4(c)
Declassify On: 20380924
Derived From: COL S-06

(U) Table of Changes

Date	Change Description	Authority
12/06/12	Created.	EDG/AED/EDB
01/31/13	Updated.	EDG/AED/EDB
09/24/13	Updated.	EDG/AED/EDB

(U) Table of Contents

1 Overview

This document is a living document, a compendium of notes on various aspects of Hive to assist current developers in tracking techniques, algorithms, and development decisions used throughout Hive and to assist future developers in understanding past development.

2 Network Resignaturing

2.1 Overview

IOC/ECG's Advanced Forensic Division (AFD) performed an analysis of Hive version 2.5 network communications to assess its likelihood of detection.The results of this analysis are found in document AFD-2012-0973-2. In summary, AFD was able to create signatures for DNS, ICMP, and TFTP triggers; found that the TCP and UDP triggers did not adhere to their respective protocol standards; and further found that the TCP and UDP triggers each had consistent packet sizes.

To address these issues, EDG modified the ICMP, TCP, and UDP triggers in Hive 2.6. The DNS and TFTP triggers were found to be problematic because each protocol is composed largely of text strings, providing virtually no fields where coded trigger packages might be hidden. Consequently, these were not addressed.

2.2 ICMP

Forensic analysts were able to discover and accurately describe the first six bytes of a common trigger within the ICMP packets. That actual trigger in its entirety is twelve bytes long and has the following format (Figure 1).

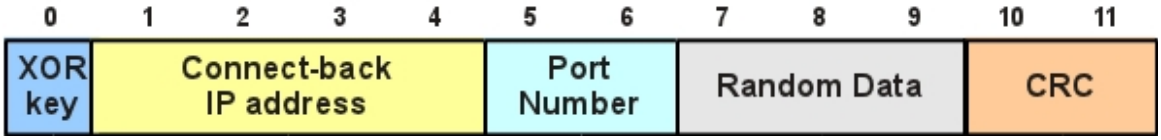

Figure 1: Hive 2.6 Common Trigger Format

This trigger was obfuscated by a simple negation. The report, however, assumed that the first byte was an XOR key for the remaining bytes in the key. As it turns out, the first byte was an opcode that was never used in Hive. Because that opcode was always zero, it negated to 0xFF.

To resignature this common trigger, the same key format was used and the first byte was randomized and then XORed with each of the remaining bytes. (Future implementations should probably use longer keys of random data.) The trigger location within the ICMP packet (bytes 4 and 5 of the timestamp) remains unchanged. The trigger is transmitted two bytes at a time in six successive ping packets.

2.3 Raw TCP and UDP

Forensic analysis of the raw TCP and UDP triggers was unable to extract a common signature for either protocol, but did note that there were identical 9-packet sequences have byte lengths of 74-74-66-70-66-466-66-66-54. While most of these lengths are typical of the protocol, what isn't typical is the unchanging 466-byte length of the trigger packet. Analysis also noted that these triggers could raise attention because they do not conform to their respective protocol specifications. However, no attempts were made to address this issue, as substantial work would be requirred to conform to upper level

protocols associated with the well known port numbers being used. For example, one would expect to see SSH traffic associated with TCP port 443, rather than raw TCP data.

Both of the raw TCP and UDP packet formats were resignatured using a similar coding strategy, but the lengths of packet are now randomized. The new packet format is shown in Figure 2 below.

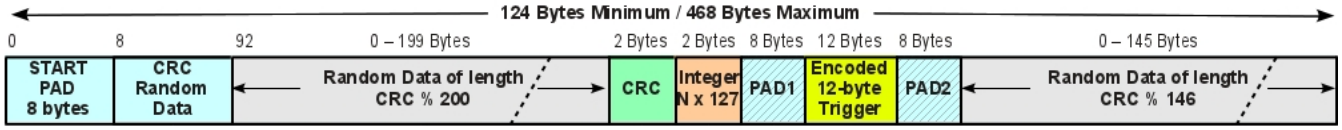

Figure 2: Hive 2.6 Raw TCP / UDP Trigger Format

Each trigger packet is built starting with a buffer sized to the maximum packet size and filled with random data. A CRC checksum is computed on a fixed length of the random data beginning after a starting pad. The CRC is then used to generate an offset from the start of the buffer where it is stored followed by a two-byte validation key (N) that is generated using a one-byte random number multiplied by 127. The common twelve-byte encoded trigger (as defined above in section 2.2) is further encoded by XORing it with random data from the buffer. The start of this random data is located before the CRC, after the start pad and computed from the CRC in combination with other parameters. The trigger is then placed in the buffer surrounded by predefined padding lengths (PAD1 and PAD2). The end of the packet is then set by computing the number of bytes to follow, once again using the CRC.

3 Self-Delete

3.1 Description

Self-delete is used to insure that any Hive implant that lays dormant (has not beaconed successfully to its designated LP or has not been triggered from a command post) for a predetermined amount of time effectively destroys itself with the only remnant being a “configuration file” (.config) and a log file (.log) left behind in /var directory. During normal operation the .log file is empty, with its last modified time indicating the time of last contact – either from a beacon or a trigger – and the .log file is non-existent. When self-delete executes, the Hive binary is deleted from the host and the log file is created with a time stamp inserted into it using the format yymmddHHMMSS. (The time stamp inserted into the file should match the last modification time of the file.)

3.2 Discrepancy Report and Analysis

Discrepancy report DR-00134-2012 was issued after Operations determined that Hive version 2.5 was self-deleting prematurely. Analysis showed that a calculation involving the current time and the file modification time used to determine the time since last contact could result in a negative number that was then cast from an integer to an unsigned long integer. This resulted in a large positive number that exceeded the delete delay and subsequently caused Hive to self-delete.

3.3 System Clock Issue

Further analysis of this issue revealed that determining when to self-delete Hive can be problematic due to the inability to accurately assess the reliability of the host's system clock. Here are three possible operational scenarios, others may exist.

1. **The host has a system clock that resets to epoch time upon reboot.** The decision to self-delete can only be determined by examining the time since last reboot and the time since last contact. The time since last contact need not be kept in non-volatile memory, as it is meaningless without a stable system clock. If the device is connected to an unstable power supply which is frequently interrupted, then Hive might never be deleted.

2. **The host has a system clock that maintains the time across reboots and may or may not be synchronized to an NTP server.** Self-deletion can be determined by comparing the time of last contact (maintained via the last modified time of the configuration file) with the current time. If a system administrator changes the time significantly from when Hive was first installed and executed, then this action could cause Hive to self-delete.

3. **The host has a system clock that resets to epoch time upon reboot, but is synchronized to an NTP server at some point after reboot.** The self-delete decision for this possibility is similar to that of scenario 2. During a short period after reboot the system clock is near epoch time, so no determination can be made until the clock is set.

Given these possibilities, there is generally no good way of knowing which environment may apply to a Hive installation. At the time of this writing, the current version of Hive (2.5.2) uses the initial time stamp of the .config file to govern the decision to self-delete. If the system clock changes significantly from the time at which Hive is first executed, then Hive may remain in the system longer than desired (the system clock gets set back from the time of original execution) or it may delete immediately (the system clock gets set forward by more than the delete delay).

3.4 Proposed Algorithm

This algorithm was proposed as one possible way of dealing with changes in the system clock, but was not implemented.

This self-delete algorithm attempts to deal with any aberrant behavior of the host's system clock. Ideally, the system clock would be set by an NTP client, but many small router devices (e.g. MikroTik) may never have NTP configured. In such cases, each reboot of the device sets the system clock back to epoch time (00:00:00, January 1, 1970). And, even if NTP is configured, there is a period of time after reboot and prior to network time synchronization when the system is running on epoch time. Consideration was given to incorporating an NTP client that would be used to create accurate time stamps, but for this revision of code it was thought to be too involved. Furthermore, if NTP is blocked by a firewall or other network device, an alternative is still needed.

There are two pieces to the algorithm: one that tries to address the system clock, and the other that tracks connection with the LP (beacon) or command post (trigger) using the configuration file modification time. This algorithm is embedded in the *check_timer* function and is called by *TriggerListen* whenever packets are available from the network to process.

3.4.1 Time Check

1. **last_time = 0** [This is the initial condition.]
   Set last_time to current time.

2. If **current time > (last_time + CHECK_INTERVAL) AND (current time - last_time) ≤ MAX_TIME_DIFF** (1 minute), then the system time is OK.
   If the time anomaly counter > 0, decrement it by 1, otherwise, **check the file configuration time**
   Set last_time to current time.

3. If **current time > (last_time + CHECK_INTERVAL) AND (current time - last_time) > MAX_TIME_DIFF**, then the system time changed.
   Increment time anomaly counter.
   Set last_time to current time.

4. If the **current time < last_time**, then the system time changed.
   Increment time anomaly counter.
   Set last_time to current time.

5. If the **time anomaly counter > TIME_ANOMALY_LIMIT**, then self delete.

The time anomaly counter gives time for anomalies to settle out before updating the configuration file. If the time anomaly counter exceeds the TIME_ANOMALY_LIMIT, it is assumed that the time is too unstable to be usable and self-delete is executed.

3.4.2 Configuration File Check

1. If **(current time - file modification time) > delete delay**,
   If there were time anomalies detected, and the counter turns zero, update the file time.
   If there were no time anomalies, self delete.

2. If **0 ≤ (current time - file modification time) < delete delay**
   Continue

3. If **(current time - file modification time) < 0**, then the system time changed.
   If the time anomaly counter is zero, update the configuration file time stamp.

3.5 Suggested Testing Methodology

The following testing methodology is used to test the proper operation of self-delete only.

Test Preparation

Testing self-delete functionality requires that the implanted target host be receiving data so that it loops through the code that listens for a trigger. The data need not be related to hive (i.e. it need not be a trigger). An easy way to accomplish this is to constantly send echo requests to the interface with an interval of 0.2 seconds or less. That is,

```
ping -i 0.2 192.168.1.1
```

where of course 192.168.1.1 is replaced with the address of the target host.

1. Determine the state of NTP on the device. If an NTP client is configured and enabled, disable it.

2. Verify that the /var/.config file does not exist. Install hived on the target host and execute. The default self-deletion time is 60 days.

3. Note the time of the system clock; this is the initiation time.

4. The /var/.config file should appear with a file date corresponding to the system time at the time it was written (which may be epoch time).

Test 1

5. Set the system clock back by more than the default delay (60 days) and verify that Hive continues to execute.

Test 2

6. Set the system clock forward of the initiation time by 59 days and verify that Hive continues to execute.

Test 3

7. Set the system clock forward of the initiation time by 61 days and verify that Hive terminates execution and leaves a ***.log*** file in the /var directory that contains a time stamp corresponding to the time of termination.

8. Reinstall Hive, removing the **.config** and **.log** files from /var.

9. Execute Hive using a delete delay of 3 minutes by adding "`-s 180`" to the execution options when Hive is started.

Test 4

10. Verify that the initiation time indicated by the last modification time on the /var/.config file matches the current system time.

Test 5

11. Verify that Hive executes for three minutes and then self-deletes, leaving behind the .config file and a .log file containing a time stamp corresponding to the time of termination and a matching file modification time.

Test 6

Test 6 is designed only for testing hive on a device that has a system clock that returns to epoch time (00:00, January 1, 1970, or there abouts) upon reboot and that also uses NTP to set the system clock.

12. Reinstall Hive, removing the **.config** and **.log** files from /var.

13. Install a startup script to make Hive execution persist across reboots. Here is a sample startup script that can be used with a MikroTik router. Place it in /etc/rc.d/run.d/S10hived for example.

```
#!/bin/bash
if [ -x /path/to/hived/hived ]; then
    /path/to/hived/hived
fi
```

14. Configure and enable the host's NTP client so that it will connect to a server that has a valid time.

15. Verify that the host synchronizes properly with the time server.

16. Reboot the device and verify that the host clock resets to epoch time and then resynchronizes to the NTP server's time after a short period.

17. Execute Hive, allowing it to run for 3 minutes. Verify that, after the 3 minutes, it is still running.

18. Reboot the host and verify that Hive restarts and continues to execute after the host's time is updated by NTP.

4 Beacon Proxy Redirection and TCP Replay

Like section 3.3 above, the system clock reseting to epoch causes other effects.

Hive beacons were designed to work with the Blot proxy (developed by Xetron). Blot looks for a tool ID embedded in the HELLO packet of an SSL session initiation. If the ID is found, then it forwards the packet to the tool-handler, otherwise it is sent to the cover server. The tool ID is embedded in the HELLO packet using the *embedData* function defined in …/polarssl/library/loki_utils.c. The SSL data structure defined in …/polarssl/include/polarssl/ssl.h is extended to include the session _checksum, tool_id, use_custom, and xor_key. The data contained within this packet is constant with the exception of a time stamp taken from the real-time clock and a few bytes of random data. A CRC checksum is computed from the entire packet and is included with the HELLO packet. When Blot receives this packet, it checks the CRC searches a list of previously seen packets for any matches. If a match is found the packet is assumed to be a TCP replay and is dropped.

If the system clock is reset to epoch time (00:00, January 1, 1970) after a reboot and the random number generator used to generate the random data that is placed into the HELLO packet is not properly seeded with pseudo-random data, then the CRC computation can produce CRCs that match previously sent HELLO packets. Such was the case discovered prior to and including Hive version 2.6.1. To eliminate this problem, the open-source Havege (Hardware Volatile Entropy Gathering and Expansion) code that is a part of the PolarSSL library was used to seed the system's *rand* function within the *irand* function found in …/polarssl/library/loki_utils.c.