National Cyber Security Centre
a part of GCHQ

Malware Analysis Report

RayInitiator & LINE VIPER

A sophisticated bootkit and user-mode capability, targeting Cisco ASA devices. A significant advancement over LINE DANCER and LINE RUNNER.

Version 1.1

October 2025

RayInitiator & LINE VIPER

A sophisticated bootkit and user-mode capability, targeting Cisco ASA devices. A significant advancement over LINE DANCER and LINE RUNNER.

Executive Summary

- RayInitiator is a persistent multi-stage bootkit which facilitates the deployment of LINE VIPER to Cisco ASA (Adaptive Security Appliance) 5500-X series devices without secure boot
- All observed targeted models have either passed their last day of support, or the last date is September 30, 2025
- LINE VIPER is a user-mode shellcode loader with associated modules
- Analysis of both LINE VIPER, RayInitiator and associated detection are contained within this report
- RayInitiator is a persistent GRand Unified Bootloader (GRUB) bootkit discovered flashed to victim devices. It survives reboots and firmware upgrades
- LINE VIPER can be tasked via two methods: WebVPN client authentication sessions over HTTPS, or via ICMP with responses over raw TCP
- LINE VIPER is loaded into memory by RayInitiator from a WebVPN client authentication request containing a partial PKCS7 certificate followed by shellcode
- LINE VIPER and RayInitiator utilise victim specific tokens, a method seen previously in LINE DANCER and LINE RUNNER
- LINE VIPER uses per-victim RSA keys for securing tasking and exfiltration via the WebVPN client authentication method
- LINE VIPER has capability to execute CLI commands, perform packet captures, bypass AAA for actor devices, suppress syslog messages, harvest user CLI commands and force a delayed reboot

Introduction

RayInitiator is a persistent multi-stage bootkit which facilitates the deployment of LINE VIPER to Cisco ASA (Adaptive Security Appliance) 5500-X series devices without secure boot.

All observed targeted models have either passed their last day of support, or the last date is September 30, 2025.

Obsolete devices introduce risk to an organisation. As such the NCSC recommends where feasible that they are replaced or upgraded. Further details can be found in our device security guidance[1].

LINE VIPER is an x64 shellcode loader that can be tasked with payloads via two methods: WebVPN client authentication sessions over HTTPS, or via ICMP with responses over raw TCP.

The deployment of LINE VIPER via a persistent bootkit, combined with a greater emphasis on defence evasion techniques, demonstrates an increase in actor sophistication and improvement in operational security compared to the ArcaneDoor campaign publicly documented in 2024.

The versions of LINE VIPER analysed in this report target Cisco ASA devices running firmware versions `9.12(4)67` and `9.14(4)24`, which at the time of the investigation were fully patched.

RayInitiator has been observed deployed to Cisco ASA models without secure boot.

Acknowledgements

The NCSC would like to thank several U.S. Government agencies, including CISA, and Cisco for the joint investigation into this malware.

[1] https://www.ncsc.gov.uk/collection/device-security-guidance/managing-deployed-devices/obsolete-products

Functionality

RayInitiator

RayInitiator is a persistent multi-stage bootkit which is flashed to the GRUB of a compromised Cisco ASA by the actor. It survives reboots and firmware upgrades.

The models of Cisco ASA devices targeted do not utilise secure boot technology and therefore there is no cryptographic verification of early boot software. These models were released in 2012, and an End of Life (EoL) notice was issued by Cisco in 2020. Newer Cisco ASA platforms include secure boot technology.

RayInitiator is responsible for installing a small handler within `lina` to execute LINE VIPER.

Note: *`lina` is the binary that implements the majority of functionality on Cisco ASA platforms.*

The bootstrapping of LINE VIPER by RayInitiator is split into multiple stages. This is summarised in the diagrams and explanations below.

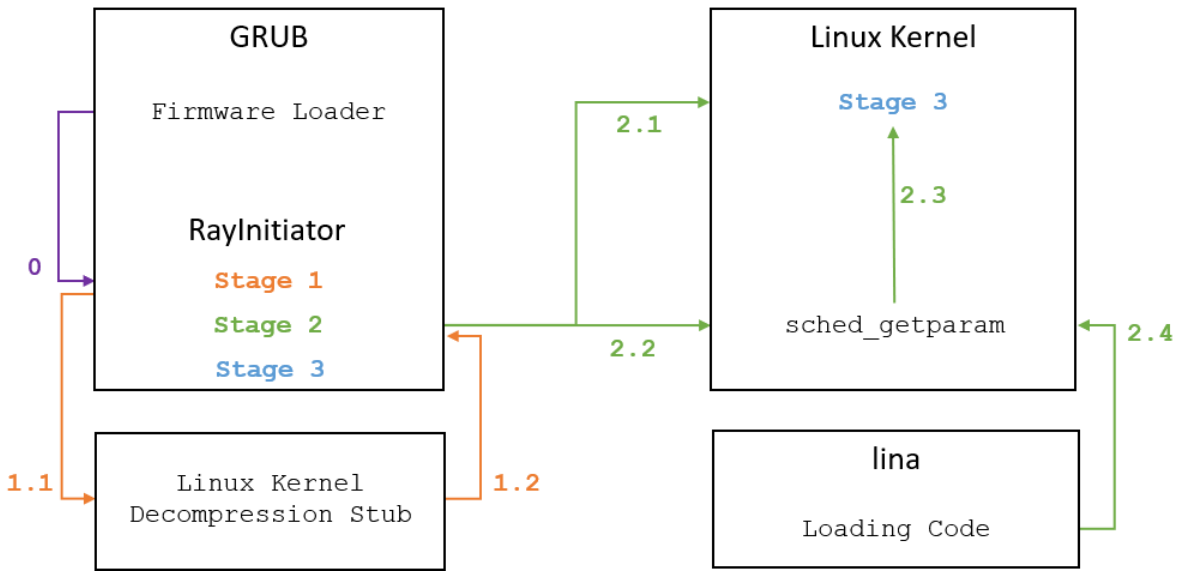

Figure 1: RayInitiator bootstrap process – part 1

RayInitiator steps 0-2, summary

0 GRUB is patched such that RayInitiator Stage 1 is called when loading the Cisco ASA firmware.

1

1.1 Search Cisco ASA loader firmware in memory for code called after the Linux kernel has been loaded.

1.2 Patch to call RayInitiator Stage 2.

2

2.1 Identify KASLR and copy RayInitiator Stage 3 into Linux kernel memory.

2.2 Identify location of `sched_getparam` within system call table.

2.3 Patch to call RayInitiator Stage 3.

2.4 `lina` calls `sched_getparam` during loading therefore triggering RayInitiator Stage 3 - Install.

Initial Execution

A compromised Cisco ASA's GRUB is patched such that RayInitiator hijacks boot execution.

This patch (step 0) hooks where the firmware is being loaded to call stage 1. Specifically, it hooks where "`Booting...\n`" is output to the console.

Stage 1

Stage 1 is responsible for searching the Cisco ASA loader in memory for code responsible for printing "`done.\nBooting the kernel`" to the console once the Linux kernel has been loaded (step 1.1) and patching it to transfer control to stage 2 (step 1.2).

Stage 1 identifies this code by first iterating the hardcoded memory region `0x400000-0x600000` to locate this string. Once found, `0x10000` is subtracted from this location and another search is performed for bytes matching the following assembly pattern:

```
48 8D 3D XX XX XX XX     lea rdi, [addr]
E8 YY YY YY YY           call [target]
```

For identified matches the `[addr]` value is compared with the location of the booting string above, to ensure the correct code is patched. The legitimate bytes are saved off for restoration in the next stage.

Finally, stage 1 restores the original code modified as part of step 0 and re-executes it such that it outputs "`Booting...\n`" to the console as expected.

Stage 2
Stage 2 is responsible for implementing the first part of the WebVPN request hook that eventually leads to the loading of LINE VIPER.

Firstly, it restores the patched code from step 1.2 and manipulates the stack, modifying the return address to enable execution of the legitimate unpatched code to output "`done.\nBooting the kernel`" after stage 2 has finished executing.

To successfully patch the Linux kernel, RayInitiator needs to identify the Kernel Address Space Layout Randomization (KASLR) base (step 2.1).

To do this it searches the previous stack frame for candidate KASLR base values. Each potential KASLR base is checked that it is `0x10000` aligned. A hardcoded fixed offset, `0x600490`, is then added to candidates. If the candidate base address is correct, the value will now point to the Linux kernel string `nmi_max_`.

Note: *The KASLR base is on the stack because at the point stage 2 is called, the first `0x40` bytes of the Linux kernel ELF header has been copied to the stack and is believed to contain the updated KASLR base. Not all Cisco ASA firmware versions support KASLR e.g., 9.12(4)67 does not whereas 9.14(4)24 does.*

The string, `nmi_max_`, is a substring of `nmi_max_handler`, which only occurs once in the kernel binary and therefore can be used to verify the correct KASLR base . The string `nmi_max_handler` is located immediately prior to the base of the system call table.

All subsequent kernel addresses and offsets from this point are adjusted using the identified KASLR base which is saved to actor-controlled memory.

The final part of step 2.1 is copying stage 3 to offset `0x300` within the Linux kernel, that ordinarily contains a large region of `nop` instructions.

Following this, in step 2.2 a fixed offset, `0x600918`, is used to locate the pointer to `sched_getparam` within the system call table, entry 143. The original pointer is saved and then overwritten with a pointer to the address of stage 3 in kernel memory (step 2.3). Such that when `sched_getparam` is called during `lina` loading (step 2.4) stage 3 gains execution.

Stage 3 has two phases, named install and deploy in this report, with install called by `lina` in step 2.4. The install phase of stage 3 ends with the below, placeholder assembly code pattern.

```
48 B8 00 00 00 00 00 00 00 00       mov rax, 0
FF E0                               jmp rax
```

Stage 2 replaces the bytes representing the null above with a pointer to the original `sched_getparam` pointer such that execution is returned to the legitimate `sched_getparam` code after stage 3's install phase has run.

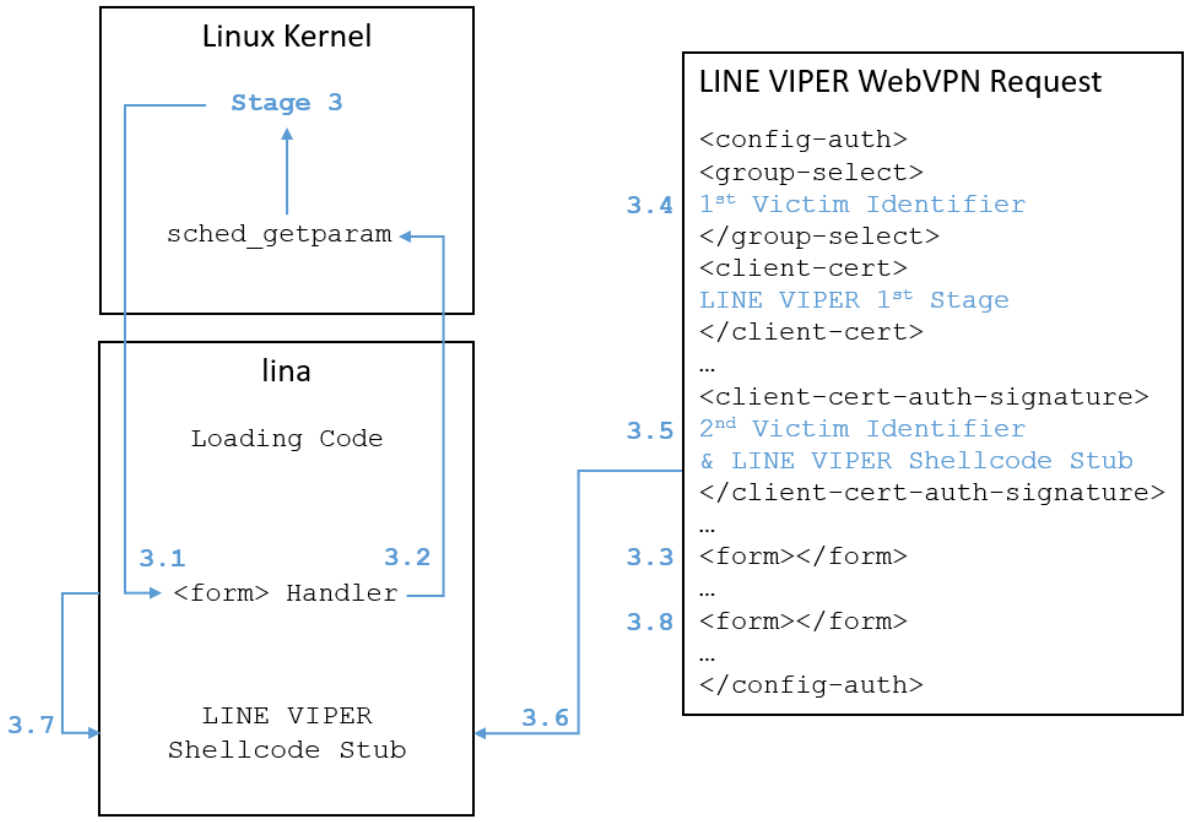

Figure 2: RayInitiator bootstrap process – part 2

Note: *WebVPN traffic on a Cisco ASA is comprised of data contained within XML tags. There is a large codebase within* `lina` *responsible for handling and processing WebVPN traffic and the associated XML data.*

RayInitiator step 3 - summary

3 Install

3.1 Search `lina` for WebVPN XML `form` element handler.

3.2 Patch to call `sched_getparam` therefore triggering RayInitiator Stage 3 – Deploy on processing of `form` element.

3 Deploy

3.3 Process the first `form` element from incoming actor WebVPN request.

3.4 Verify `<group-select>` for first victim identifier.

3.5 Verify an XML element, e.g., `<client-cert-auth-signature>`, for second victim identifier followed by LINE VIPER shellcode stub.

3.6 Copy shellcode stub to `form` local data area within `lina` and mark executable.

3.7 Overwrite `form` handler to call shellcode on processing of second `form` element.

3.8 Process second `form` element.

Stage 3

Stage 3 has two separate phases, install and deploy, with the first called by `lina` as part of step 2.4 and is responsible for installing a hook into the processing of network traffic containing a specific WebVPN XML element, `form`.

The deploy phase is responsible for parsing and executing LINE VIPER from a WebVPN request.

Install phase

During this phase, RayInitiator searches `lina` for the table entry that handles the WebVPN XML `form` element (step 3.1). This is so it can be modified with a pointer to stage 3 and allow the deploy phase to be run. Details of this phase are outlined below.

When operating in the install phase, stage 3 is running in the context of a system call and has been called with the arguments of a legitimate call to `sched_getparam`.

One of the arguments passed is a `lina` memory address. RayInitiator takes this address and subtracts `0x8000000` to provide a region of memory to search. It searches this region of memory for the string `client-cert-fail`. From the location that string is found, `0x2000000` is added to then search for a pointer to the above string. If this pointer is found, this indicates that it is within the WebVPN XML element parsing table.

Note: *The WebVPN XML element parsing table is a region of the `lina` data section which is used by WebVPN when processing requests. The table contains entries for each of the XML tags which are supported by WebVPN, the entries are `0x260` bytes long and begin with a pointer to the string for which the XML element table entry is for e.g., `form`. At offset 0x40 from the base is a pointer to the function which handles data for the specific tag. The table entry is also used to store tag-specific data such as pointers and request data.*

Stage 3 uses the identified memory address above within the WebVPN XML element parsing table, subtracts `0x2140` to obtain the beginning of the table, and adds `0x4E60` to get the end. This table is then searched for the `form` element entry.

When it has located the entry, the handler function is overwritten at offset `0x40`, with a pointer to `sched_getparam` (step 3.2). Ordinarily there is no handler for the `form` element entry.

Therefore, when processing a LINE VIPER WebVPN request the first `form` element encountered causes a call to the `sched_getparam` system call entry which instead executes stage 3 again but this time for the deploy phase.

Note: *The reason the string `client-cert-fail` is used to identify the table and not the `form` string directly is likely due to the uniqueness of the string and the fact it only has one pointer to it within `lina`.*

Deploy phase

The deploy phase of stage 3 is responsible for parsing and executing LINE VIPER from a WebVPN request containing a `form` element. When the stage 3 deploy phase is running it is within the context of a WebVPN XML element handler.

Crucially because it is called in this context, the argument passed is an address of a structure containing pointers to the WebVPN XML element data currently being processed.

To check for a LINE VIPER request, the `<group-select>` element is verified to ensure it starts with a hard-coded, victim specific, 8-byte ASCII string (step 3.4).

This is then followed by iterating the other XML elements in the request looking for one that starts with a second hard-coded victim specific 8-byte hexadecimal value. Immediately following this value will be the LINE VIPER shellcode stub (step 3.5). At this point the WebVPN code in `lina` has Base64 decoded relevant XML element data.

It is known from observed traffic that the LINE VIPER shellcode stub is located within the `<client-cert-auth-signature>` element, but the actor has flexibility as to which element to include this shellcode within because all elements are checked for the prefixed 8-byte hexadecimal value.

Following this, stage 3 starts by identifying the location of the `form` entry within the WebVPN XML element parsing table.

To do this it iterates the general-purpose registers used for the local variables of the calling function for the handler i.e. a legitimate `lina` function.

It checks each for a pointer to the base of the `form` element entry of the WebVPN XML element parsing table. It confirms the entry by dereferencing and comparing the handler function against the pointer to `sched_getparam`.

Once this table entry has been found, the LINE VIPER shellcode stub is copied to the request data section of the `form` element entry of the WebVPN XML Element parsing table, step 3.6.

This region of memory is not executable as it is in the `.data` section of `lina`. Stage 3 does a direct system call to `mprotect`, using a fixed offset to system call 10, to mark the encompassing page as executable.

Finally, stage 3 overwrites the `form` handler again to instead point to the LINE VIPER shellcode stub (step 3.7). Therefore, when the second `form` element observed in network traffic is processed, it triggers the installation of LINE VIPER via the shellcode stub (step 3.8).

LINE VIPER

LINE VIPER is composed of a core shellcode loader, split into numerous sections of x64 shellcode. It also encompasses modular capability deployed to it as shellcode tasking payloads. The core functionality is more comprehensive than in LINE DANCER, with a large proportion of code dedicated to handling secure encryption for communication between the actor and victim.

Note: *Instances of LINE VIPER and RayInitiator are victim specific, as such hashes are not provided.*

Capability is once again implemented within shellcode tasking payloads that extensively rely on calling and hooking legitimate `lina` code.

LINE VIPER contains capability to execute CLI commands, perform packet captures, bypass AAA, suppress syslog messages, harvest user command-line interface (CLI) commands and force a delayed reboot.

Bootstrap

LINE VIPER has a series of steps to bootstrap itself into device memory. The shellcode steps are chained together.

Initial execution

LINE VIPER is deployed to devices compromised with RayInitiator via a tasking packet containing Base64-encoded shellcode disguised as a certificate within a WebVPN client authentication request.

For observed deployments initial execution is via a `<client-cert-auth-signature>` element containing a Base64-encoded shellcode stub.

```
<client-cert-auth-signature hash-algorithm-chosen="sha512">
```

Figure 3: XML element containing initial shellcode

As discussed in the RayInitiator stage 3 section, this LINE VIPER shellcode is executed in the context of a WebVPN XML element handler (see Deploy mode).

As such, when called it is passed an argument that is the address of a structure that has pointers to the WebVPN XML element data currently being processed i.e. data inside other XML elements that contain the core of the LINE VIPER malware.

```
<client-cert cert-format="pkcs7">
```

Figure 4: XML element containing partial certificate and main shellcode

The Base64-decoded bytes from the above element contain `0x80` bytes of a legitimate VeriSign certificate, followed by the remaining LINE VIPER code.

The serial number of the partial certificate is: `3037644167568058970164719475676101450`.

The initial shellcode stub skips past these bytes and begins execution of the main shellcode.

The inclusion of the beginning of a certificate is believed to be an attempt to make the traffic blend in, as when Base64-encoded the data starts with the familiar `MII` of a certificate.

XOR obfuscation

The shellcode from within the certificate XML element in Figure 4 begins with a small stub to XOR-deobfuscate the rest of itself.

This stub XORs every obfuscated byte with the entirety of a 32-byte key and is therefore equivalent to a single-byte XOR.

This 32-byte XOR key is unique per victim and is taken from a `group-access` XML element at offset `0x70` as raw bytes.

```
<group-access>
```

Figure 5: XML element containing 32-byte XOR key

Staging

After deobfuscation LINE VIPER has three sections of code as part of staging that do not execute sequentially.

The first section is responsible for the setup and loading of the other sections of the LINE VIPER malware into memory.

It starts by wiping the LINE VIPER shellcode stub and RayInitiator stage 3 `form` element handler then restores memory permissions.

Following this it copies the second section to an area of the `lina .text` section. This area is the beginning of a large, legitimate and unused `lina` function used by the actor as a code cave.

Note: *A code cave is an area of an executable binary which is not used under normal execution. It is therefore safe for malware authors to overwrite legitimate code with their malicious code and redirect execution to it.*

It also copies the third section to the `.data` section of `lina` which is initially marked as read/write.

Finally, the first section hooks `lina` to intercept handling of VPN client authentication requests which it redirects to the code cave containing the second section of LINE VIPER.

The second section is responsible for verifying whether tasking requests contain a 32-byte victim token.

This token should be present at the start of the device type XML attribute and is victim specific.

```
<device-id computer-name="intel-PC" device-type=
```

Figure 6: XML attribute containing the 32-byte victim token

If the request is verified to contain the victim token, then the area of memory containing the third section of LINE VIPER, responsible for parsing tasking packets is temporarily marked executable, executed, and then restored to read/write afterwards. This technique is commonly referred to as a gargoyle.

The third section contains the code for handling tasking and exfiltration over VPN client authentication requests/responses covered in the Communication – VPN client authentication method section of this report.

Capability

A variety of deployed modules have been identified as part of LINE VIPER compromises. Some of this capability is new and some has been expanded over LINE DANCER. Either of the two tasking methods discussed later in this report could be used to deploy the observed modules.

CLI commands

LINE VIPER has the ability to grant itself level 15 privilege and execute CLI commands, for example:

- `show vpn-sessiondb anyconnect`

Packet captures

LINE VIPER has the ability to perform packet captures hidden from view of administrative commands i.e. packet captures launched by LINE VIPER do not appear if an administrator runs `show capture`. The following protocols have been observed being captured: RADIUS, LDAP and TACACS.

For this functionality the actor uses payloads to:

- start captures, e.g., `capture <name> buffer 10066410 interface <interface> match <filter>`
- stop captures, e.g., `no capture <name>`
- show captures, e.g., `show capture`
- save captures by exfiltrating the capture buffer over the aforementioned TCP method.

LINE VIPER hides packet captures by overwriting and restoring an internal `lina` structure with its own as required during payload execution therefore maintaining its own separate list.

AAA device bypass

LINE VIPER has the ability to bypass VPN Authentication, Authorization and Accounting (AAA) for actor devices.

This is achieved by LINE VIPER maintaining a table of actor utilised device IDs that bypass AAA checks.

The actor device ID, `[DEVICE_ID]`, would appear as the unique ID in the following blank `device-id` WebVPN XML element:

```
<?xml version="1.0" encoding="UTF-8"?>
<config-auth client="vpn" type="init" aggregate-auth-
version="2">
<version who="vpn"></version>
<device-id computer-name="" device-type="" platform-version=""
unique-id="[DEVICE_ID]"></device-id>
```

Figure 7: Device ID XML Element

To implement this functionality a number of `lina` functions are patched to check incoming requests against an actor-controlled device ID table. Each entry in this table is `0x80` bytes, `0x40` bytes for the device ID and `0x40` bytes for what appears to be an account.

In particular the following patches have been observed:

- Patch to create a new internal `lina` structure believed to relate to AAA authentication.
- Patch such that a request is handled as if the client had presented a certificate.
- Patch to bypass two checks relating to whether a certificate is required for a tunnel group.
- Patch to bypass the copying to syslog within Dynamic Access Policy (DAP) handling.

CLI command harvester

LINE VIPER has the ability to harvest commands entered on the Cisco CLI.

To achieve this two functions handling CLI commands are hooked, in two locations each, at the point when they are about to write to the syslog. These hooks additionally copy the CLI command to actor-controlled memory.

These syslog IDs and messages are shown in the table below.

Syslog ID	Message Format
111008	`%d: User '%s' executed the '%s' command.`
111009	`%d: User '%s' executed cmd: %s\n`

These messages when formatted are prefixed with their severity levels, `5` and `7` respectively, i.e., `5111008` and `7111009`.

Force delayed reboot

LINE VIPER has a module that forces a delayed reboot after a set period, unless it is cancelled by setting a cancel flag.

The module sleeps in a loop of `1800` second intervals, each time checking whether a flag is set, if so, the reboot is cancelled. This sleep period has been observed ranging between one and ten hours before rebooting.

Defence evasion

LINE VIPER features a number of design decisions and capability attempting to prevent identification of compromise.

Syslog suppression

For LINE DANCER the actor had payloads to completely disable and re-enable syslog during operations.

In LINE VIPER this capability has been improved where instead specific syslog IDs and messages are supressed. This means that whilst it is not as immediately obvious that syslog is being tampered with, it is still identifiable.

These syslog IDs and messages are shown in the table below and are believed to relate to actor follow-on activity.

ID	Message Format
302013	Built *{inbound | outbound}[Probe]* TCP connection *connection_id* for *interface*:*real-address/real-port* (*(mapped-address/mapped-port))idfw_user* to *interface*:*real-address/real-port* (*mapped-address/mapped-port)inside_idfw_and_sg_info id_port_num rx_ring_num* [(*user*)]
302014	Teardown [Probe]TCP connection connection_id for interface:real_address/real_portidfw_user to interface:real_address/real_portidfw_user duration hh:mm:ss bytes bytes reason_stringteardown_initiatorinitiator port_num rx_ring_num [(user)]
609002	Teardown local-host *zone_name*:*ip_address* duration *time*
710005	{TCP|UDP|SCTP} request discarded from source_address/source_port to interface_name:dest_address/service

Additionally, LINE VIPER hooks a syslog function to check for the string `group` and if found skips a section of further syslog code. It is unclear whether this is an attempt to suppress syslog messages containing `group` or not.

Memory permissions

As part of its initialisation LINE VIPER needs to temporarily modify memory permissions for locations within the `.text` section of `lina` so that they are writeable.

Previously, in LINE DANCER this was performed on small sections of memory which led to the 'split lina' effect, observable from the output of the following CLI command.

```
show memory region | include lina
```

Figure 8: CLI command for checking 'split lina' effect

For LINE VIPER the actor has fixed this issue by instead modifying the memory permissions of the entire code section starting at `cs:0` up to the length at the end of the `.gcc_except_table` section up to the first full page (4096 bytes).

This means that memory regions for `lina` will not be split. The original memory protections are restored after modifications are made.

LINE VIPER also modifies memory permissions for specific data sections to allow them to be writeable/executable. Unlike for the code section however these modifications are not made to the entire region. Despite this, once the original memory protections are restored these regions are merged by the Linux kernel back together so the 'split lina' effect is not observable.

Anti-forensics

LINE VIPER performs a number of modifications to `lina` to prevent forensic collection or identification of compromise for certain CLI commands.

- Modified signal handler for when a core dump is triggered such that instead the device immediately reboots.
- Modified `copy` command function pointer to an actor's function that checks for `system:/text` and reboots the device if found. This comparison accounts for additional characters before `text` and for auto-completion of the command, e.g., `tex`.
- Modified `verify` command to perform the same check as above and if found, returns the hash of `text` section of legitimate firmware instead, either MD5 or SHA-512 depending which was requested.

Communications

Two methods for tasking LINE VIPER have been identified. The first within VPN client authentication sessions via HTTPS, and the second via ICMP with responses sent via raw TCP. These methods are used to deploy the capability of LINE VIPER or exfiltrate collected data.

Note: *The ICMP method is believed to be deployed via the VPN client authentication method.*

VPN client authentication method

Overview

The actor uses secure cryptography to protect command-and-control, including using victim specific encryption keys. This prevents analysis of subsequent tasking and exfiltration.

This tasking method involves two steps, both with a request and response. The first step is responsible for sharing randomly-generated key material. The second step uses this key material to de/encrypt tasking and exfiltration.

Key exchange

For an initial key exchange packet, after the first victim token is verified the data that follows is Base64-decoded.

If the decoded data starts with another victim token, then this signifies that it is part of the key material sharing step.

For this step the data following the second victim token is RSA decrypted using a per-victim RSA public key.

After RSA decryption a third victim token is verified. If correctly derived, this confirms that the request to share key material was correctly formed.

Note: *Therefore, whilst an RSA public key is utilised, it is simply used to derive and verify a pre-shared, victim specific, token. This ensures that tasking is from the actor as they have the corresponding RSA private key.*

If all victim tokens match, 128 random bytes are generated by LINE VIPER and stored in memory.

These 128 bytes are split and used in the second step as follows: 32-byte pre-decryption challenge, 32-byte post-decryption challenge, 16-byte decryption IV, 16-byte encryption IV and 32-byte de/encryption key. LINE VIPER can store 48 sets of the above challenges and key material.

These random bytes are RSA encrypted using the same public key above, and then Base64-encoded. The result is then formatted within a message XML element and returned as part of the response.

```
<config-auth client="vpn" type="auth-request" aggregate-auth-
      version="2"><client-cert-request></client-cert-
                  request><message>
```

Figure 9: XML element containing RSA encrypted key material

Tasking

The second step, i.e. tasking, starts the same but is differentiated based on the absence of the second victim token. An example of a tasking request and a tasking response can be found in Appendices 2 and 3.

Tasking starts with one of the pre-decryption challenges from above which indicates the key material to use. If none of the challenges are verified then the request is discarded.

After this challenge the rest of the data is decrypted using AES-CBC-256 with the corresponding decryption IV and key. Decrypted data is prefixed with the post-decryption challenge for verification. This confirms that the request was generated using the necessary RSA private key.

Once confirmed, the remaining AES decrypted data is copied into memory and executed. After tasking execution this payload memory, of length `0x2000` regardless of actual length, is then cleared.

The result of the completed tasking is then AES encrypted using the same key. However, the encryption IV this is overwritten with the decryption IV prior to use. The reason for this is unknown. Finally, the encrypted response is Base64-encoded, and again formatted within a message XML element.

This ensures that even if actor payloads and exfiltration are intercepted, they cannot be decrypted.

ICMP & TCP method

Overview

A second tasking method, via ICMP, has been observed. Responses to tasking, i.e. exfiltration, via this method are sent in raw TCP packets.

Crucially, in observed operations, the ICMP tasking is not sent to the wide area network (WAN) interface of the device and is instead tunnelled in an established VPN session and sent to a local area network (LAN) interface of the Cisco ASA.

However, the TCP responses originate from the WAN interface of the device and are sent to the VPN LAN IP of the actor's device in the VPN pool.

Tasking

ICMP payloads are sent as Echo Requests where the ICMP data starts with the same first victim token. This is followed by a length and the payload data. Tasking and exfiltration via this method contain no additional encryption beyond that provided by the VPN tunnel.

However, the connection between the VPN LAN IP of the actor and the Cisco ASA occurs entirely within the local network, thus observation of this C2 channel is difficult using traditional network observation methods.

Payloads

For observed ICMP tasking payloads sent to LINE VIPER there is consistent actor boilerplate code.

A stream buffer channel, `sbuf`, is used to manage the output of `lina` functions called as part of tasking. When LINE VIPER is ready to retrieve the data for exfiltration, IOCTL `0x12001` is called on the channel to identify the length followed by calling IOCTL `0x12008` to obtain the data itself.

This length and data are then passed to a new actor process. This process has the name `Unicorn Proxy Thread`, which is a legitimate name found in `lina` ordinarily used to handle WebVPN traffic.

Each ICMP tasking payload contains the IP address and TCP destination port which the tasking response should be sent to. Observed destination ports used are random high ports, >`60000`.

This actor process calls into multiple `lina` functions. Whilst the code has not been fully analysed an overview is as follows:

- A channel is created using the following format string: `tcp/CONNECT/%d/%A/%d`
- Two IOCTLs, `0x1901A` and `0x19020`, are called that appear to relate to setting an internal `lina` option `TCPTO`, believed to be used as a timeout. The value set is `7232` milliseconds
- An IOCTL, `0x19021`, is called that appears to relate to setting the option `TCPWINDOWSIZE`, the value set is `16384`. Observed traffic has also had a value of `32768`, it is unclear if this is set or overridden elsewhere by `lina`
- The exfiltration passed to this process is then sent in subsequent TCP packets
- Finally, a carriage-return and new-line, `\r\n`, are sent in a final TCP packet

Conclusion

RayInitiator and LINE VIPER constitute a significant evolution over malware from the previous campaign, LINE DANCER and LINE RUNNER. Both show an increase in sophistication and an improvement of operational security.

RayInitiator is a complex GRUB bootkit giving the actor persistent access. The understanding of the Cisco ASA platform required to implement this capability demonstrates an actor with dedicated expertise. It also indicates a confidence in their operations, given both the risk involved and remediation required to reflash an SPI chip is non-trivial.

LINE VIPER is a shellcode payload loader and modular capability, with particular attention paid to anti-forensics and protection of its command-and-control communication. The modules deployable to LINE VIPER are broad in their capability. This highlights an actor who learns from and responds to the techniques used by network defenders.

Detection

Detection mechanisms such as rules, signatures and scripts are licensed under the terms of the Open Government Licence v3.0 except where otherwise stated.

Overview

Cisco has released a blog post detailing this campaign, with sections on detection and remediation, including the release of patches. This guidance should be followed in the first instance.

If you discover the presence of LINE VIPER or RayInitiator on your device, please report it to the respective cyber security centre or agency in your jurisdiction. In the UK that is the National Cyber Security Centre (NCSC).

For detection it is important to emphasise that:

- the actor employs several anti-forensic methods
- whilst a device might be compromised with RayInitiator, LINE VIPER might not necessarily be deployed.
- If a core dump is attempted before upgrading to Cisco's patched firmware and the device immediately reboots without producing a core dump, then this could indicate that LINE VIPER was deployed.
- If instead however a core dump was successful, or attempted again after reboot, then it should be investigated for evidence of RayInitiator; in particular, the modification to the WebVPN XML element handler table as discussed in the report. A script that attempts to detect this is in Appendix 1.

Rules and signatures

Description	Detects RayInitiator GRUB bootkit stage 1 code that searches for the 'Booting the kernel' string.
Precision	No false positives have been identified.
Rule type	YARA

```
rule RayInitiator_stage_1_search_for_booting_kernel_string {
    meta:
        author = "NCSC"
        description = "Detects RayInitiator GRUB bootkit stage 1
code that searches for the 'Booting the kernel' string."
        date = "2025/09/25"
    strings:
        $ = {BB 00 00 40 00 43 81 FB 00 00 60 00 0F 87 AB 00 00 00
8B 3B 81 FF 64 6F 6E 65 75 E9 83 C3 04 8B 3B 81 FF 2E 0A 42 6F 75
DC 83 C3 04 8B 3B 81 FF 6F 74 69 6E 75 CF 83 C3 04 8B 3B 81 FF 67
20 74 68 75 C2 83 C3 04 8B 3B 81 FF 65 20 6B 65 75 B5 83 C3 04 8B
3B 81 FF 72 6E 65 6C 75 A8 83 EB 14}
    condition:
        any of them
}
```

Description	Detects RayInitiator GRUB bootkit stage 2 code that identifies the Linux kernel syscall table.
Precision	No false positives have been identified.
Rule type	YARA

```
rule RayInitiator_stage_2_identify_syscall_table {
    meta:
        author = "NCSC"
        description = "Detects RayInitiator GRUB bootkit stage 2
code that identifies the Linux kernel syscall table."
        date = "2025/09/25"
    strings:
        $ = {49 89 E0 48 83 F8 30 0F 84 70 00 00 00 49 01 C0 49 8B
10 48 83 C0 08 66 85 D2 75 E4 BF ?? ?? 60 00 48 8B 3C 17 48 BE 6E
6D 69 5F 6D 61 78 5F}
    condition:
        any of them
}
```

Description	Detects RayInitiator GRUB bootkit stage 3 install phase code that searches for the 'client-cert-fail' string.
Precision	No false positives have been identified.
Rule type	YARA

```
rule
RayInitiator_stage_3_install_phase_search_for_client_cert_fail_str
ing {
    meta:
        author = "NCSC"
        description = "Detects RayInitiator GRUB bootkit stage 3
install phase code that searches for the 'client-cert-fail'
string."
        date = "2025/09/25"
    strings:
        $ = {48 81 EE 00 00 00 08 48 B8 63 6C 69 65 6E 74 2D 63 49
B8 65 72 74 2D 66 61 69 6C 48 FF C6 48 39 D6 0F 87 D2 00 00 00 48
8B 3E 48 39 C7}
    condition:
        any of them
}
```

Description	Detects RayInitiator GRUB bootkit stage 3 deploy phase code that copies LINE VIPER shellcode stub and marks executable.
Precision	No false positives have been identified.
Rule type	YARA

```
rule
RayInitiator_stage_3_deploy_phase_copy_LINE_VIPER_shellcode_stub {
    meta:
        author = "NCSC"
        description = "Detects RayInitiator GRUB bootkit stage 3
deploy phase code that copies LINE VIPER shellcode stub and marks
executable."
        date = "2025/09/25"
    strings:
        $ = {48 89 FA 48 83 C7 40 4C 89 CE B9 D0 01 00 00 F3 A4 48
89 D7 48 83 C7 40 48 89 3A 48 C1 EF 0C 48 C1 E7 0C BA 07 00 00 00
48 C7 C6 00 20 00 00}
    condition:
        any of them
}
```

Description	Detects LINE VIPER Cisco ASA malware code as part of a shellcode deobfuscation routine.
Precision	No false positives have been identified.
Rule type	YARA

```
rule LINE_VIPER_shellcode_deobfuscation_routine {
    meta:
        author = "NCSC"
        description = "Detects LINE VIPER Cisco ASA malware code
as part of a shellcode deobfuscation routine."
        date = "2025/09/25"
    strings:
        $ = {48 8B 7F 08 48 8D 5F 70 49 C7 C1 00 18 00 00 49 C7 C0
20 00 00 00 48 89 DF 8A 01 32 07 48 FF C7 41 FF C8 4D 85 C0 75 F3
88 01 48 FF C1 41 FF C9 4D 85 C9 75 DA}
        $ =
"SIt/CEiNX3BJx8EAGAAAScfAIAAAAEiJ34oBMgdI/8dB/8hNhcB184gBSP/BQf/JT
YXJdd"
        $ =
"iLfwhIjV9wScfBABgAAEnHwCAAAABIid+KATIHSP/HQf/ITYXAdfOIAUj/wUH/yU2
FyXXa"
        $ =
"Ii38ISI1fcEnHwQAYAABJx8AgAAAASInfigEyB0j/x0H/yE2FwHXziAFI/8FB/8lN
hcl12"
    condition:
        any of them
}
```

Description	Detects LINE VIPER Cisco ASA malware code as part of shellcode initial execution.
Precision	No false positives have been identified.
Rule type	YARA

```
rule LINE_VIPER_shellcode_initial_execution {
    meta:
        author = "NCSC"
        description = "Detects LINE VIPER Cisco ASA malware code
as part of shellcode initial execution."
        date = "2025/09/25"
    strings:
        $ = {48 8D B7 80 00 00 00 BA 00 20 00 00 [19] 48 C7 C6 00
90 00 00 BA 07 00 00 00}
        $ = /SI23gAAAALoAIAAA[A-Za-z0-9+\/]{26}jHxgCQAAC6BwAAA/
        $ = /iNt4AAAAC6ACAAA[A-Za-z0-9+\/]{26}Ix8YAkAAAugcAAA/
        $ = /IjbeAAAAAugAgAA[A-Za-z0-9+\/]{26}SMfGAJAAALoHAAAA/
    condition:
        any of them
}
```

Description	Detects LINE VIPER Cisco ASA malware code as part of RSA encrypted random AES key generation.
Precision	No false positives have been identified.
Rule type	YARA

```
rule LINE_VIPER_rsa_encrypted_random_aes_key_generation {
    meta:
        author = "NCSC"
        description = "Detects LINE VIPER Cisco ASA malware code
as part of RSA encrypted random AES key generation."
        date = "2025/09/25"
    strings:
        $ = {48 31 C0 49 89 06 49 89 46 08 49 83 C6 10 49 83 ED 10
4D 85 ED 75 D8 BF 30 00 00 00}
        $ = {0F 85 57 01 00 00 49 8B 44 24 08 48 83 F8 2F 7C 33 41
BD F0 02 00 00 4D 8D 74 24 10 49 8B 3E}
        $ = {85 C0 0F 8E EE 00 00 00 41 BD F0 02 00 00 4D 8D 7C 24
10 49 8B 3F 48 85 FF 74 0D 49 83 C7 10 49 83 ED 10 4D 85 ED 75 EB
4D 89 37 BF 70 00 00 00}
        $ = {48 85 C0 0F 84 3F 00 00 00 48 89 45 B0 BF 80 00 00 00
4C 89 EE 48 89 C2 48 8B 4D A8 41 B8 01 00 00 00}
    condition:
        3 of them
}
```

Description	Detects LINE VIPER Cisco ASA malware code as part of AES encrypted tasking and exfiltration.
Precision	No false positives have been identified.
Rule type	YARA

```
rule LINE_VIPER_aes_encrypted_tasking_and_exfiltration {
    meta:
        author = "NCSC"
        description = "Detects LINE VIPER Cisco ASA malware code
as part of AES encrypted tasking and exfiltration."
        date = "2025/09/25"
    strings:
        $ = {48 31 C0 48 89 45 D8 49 89 FC 49 89 F5 49 89 D6 48 8B
47 08 48 89 45 B8 48 8D 40 40 48 89 45 E0 48 8D 70 E0 48 89 75 B0
48 8D 78 F0 48 89 7D E8 BA 10 00 00 00}
        $ = {48 85 C0 0F 84 EA 00 00 00 48 89 45 A8 4C 89 EF 48 89
C6 4C 89 F2 48 8B 4D A0 4C 8B 45 B0 4D 31 C9}
        $ = {48 85 C0 0F 84 82 00 00 00 49 89 C7 48 8B 7D E0 BE 00
01 00 00 48 8B 55 A0}
        $ = {48 8B 7D D0 49 83 C7 10 49 C1 EF 04 49 C1 E7 04 4C 89
FE 48 8D 55 D8}
    condition:
        3 of them
}
```

Description	Detects LINE VIPER Cisco ASA malware code as part of ICMP tasking shellcode payloads.
Precision	No false positives have been identified.
Rule type	YARA

```
rule LINE_VIPER_icmp_tasking_shellcode_payloads {
    meta:
        author = "NCSC"
        description = "Detects LINE VIPER Cisco ASA malware code
as part of ICMP tasking shellcode payloads."
        date = "2025/09/25"
    strings:
        $ = {55 53 41 54 41 55 41 56 41 57 48 89 E5 48 83 EC 60 48
31 C0 B9 07 00 00 00 48 8D 7D A8 F3 48 AB BF 01 00 00 00 BE 30 00
00 00}
        $ = {49 89 C7 48 C7 C2 38 DF FF FF 64 48 8B 0A 48 8B 99 00
01 00 00 48 89 81 00 01 00 00}
        $ = {49 8B 47 10 48 8D 55 B0 BE 01 20 01 00 4C 89 FF FF 90
90 00 00 00 48 8B 7D B0 48 85 FF 0F 84 3C 00 00 00}
        $ = {49 8B 47 10 BE 08 20 01 00 4C 89 FF 48 8D 55 A8 FF 90
90 00 00 00 48 8B 7D B0 49 89 7E 20 48 8B 7D A8 49 89 7E 28}
    condition:
        3 of them
}
```

Appendices

Appendix 1 – RayInitiator coredump detection Python script

The following Python script can be used to detect evidence of RayInitiator within a Cisco ASA coredump. It is provided as is and the NCSC cannot guarantee complete detection coverage. It has been tested on coredumps from affected versions of 9.12* and 9.14*. The script has not been tested on every version of the Cisco ASA firmware. It works by searching for the actor's hook into `lina`.

```
import sys
import gzip
import struct

def detect_rayinitiator(core):
    #check coredump starts with ELF header
    magic = struct.unpack("I", core[:4])[0]
    if magic != 0x464C457F:
        print("[ERROR] Core does not begin with expected ELF magic")
        return

    #obtain offset to program header table
    phoff = struct.unpack("Q", core[0x20:0x28])[0]

    #check that the program header entry size is 0x38
    phentsize = struct.unpack("H", core[0x36:0x38])[0]
    if phentsize != 0x38:
        print("[ERROR] Unexpected phentsize - script only handles size
0x38")
        return

    #obtain the second segment and check it is RX as expected
    rx_segment_loc = phoff + (phentsize * 1)
    rx_segment = core[rx_segment_loc:rx_segment_loc+phentsize]

    flags = struct.unpack("I", rx_segment[4:8])[0]

    if flags != 5:
        print("[ERROR] Unexpected flags for segment 2 - expected 5 - RX")
        return

    #obtain the third segment and check it is RW as expected
    offset = struct.unpack("Q", rx_segment[8:16])[0]
    size = struct.unpack("Q", rx_segment[32:40])[0]
    rx_data = core[offset:offset+size]
    rx_virtual = struct.unpack("Q", rx_segment[16:24])[0]

    rw_segment_loc = phoff + (phentsize * 2)
    rw_segment = core[rw_segment_loc:rw_segment_loc+phentsize]

    flags = struct.unpack("I", rw_segment[4:8])[0]

    if flags != 6:
        print("[ERROR] Unexpected flags for segment 3 - expected 6 - RW")
        return
```

```
    offset = struct.unpack("Q", rw_segment[8:16])[0]
    size = struct.unpack("Q", rw_segment[32:40])[0]
    rw_data = core[offset:offset+size]
    rw_virtual = struct.unpack("Q", rw_segment[16:24])[0]

    #look for the 'client-cert-fail' string within the RX segment
    crib = b"client-cert-fail"
    loc = rx_data.find(crib)
    if loc == -1:
        print("[ERROR] Could not find crib 'client-cert-fail'")
        return

    #look for a pointer to this string within the RW segment
    loc = loc + rx_virtual
    ptr = rw_data.find(struct.pack("Q", loc))
    if ptr == -1:
        print("[ERROR] Could not find pointer to crib string")
        return

    #obtain the element table by subtracting and adding fixed length values
(as used by RayInitiator)
    element_table_start = ptr-0x2140
    element_table = rw_data[element_table_start:element_table_start+0x4E60]

    #iterate over the element handler table and check for the form entry
    form_entry = b""
    for i in range(0, len(element_table), 0x260):
        element_entry = element_table[i:i+0x260]

        #obtain pointer to element name
        element_name_loc = struct.unpack("Q", element_entry[0:8])[0]
        element_name_loc = element_name_loc - rx_virtual
        if element_name_loc < 0:
            print("[ERROR] Reached end of element handler table without
finding form entry")
            return

        #obtain element name from the RX segment
        element_name = rx_data[element_name_loc:]
        element_name = element_name[:element_name.find(b"\x00")]

        #check if the form element entry
        if element_name == b"form":
            form_entry = element_entry
            break

    #check whether the form entry was identified
    if form_entry == b"":
        print("[ERROR] Reached end of element handler table without finding
form entry")
        return

    #obtain the form handler pointer
    form_handler = struct.unpack("Q", form_entry[0x40:0x48])[0]

    #if the form handler is 0 then RayInitiator has not been detected
    if form_handler == 0:
        print("[RESULT] No malicious RayInitiator handler detected")
    else:
```

```
        print("[RESULT] Malicious RayInitiator handler detected: 0x%X" %
form_handler)

try:
    #gzip decompress provided core and attempt detection of RayInitiator
    core = gzip.open(sys.argv[1], 'rb').read()
    detect_rayinitiator(core)
except:
    print("[ERROR] Provided coredump failed to extract - is it corrupt?")
```

Appendix 2 – VPN client authentication tasking request

An example actor tasking request can be seen below.

```
POST / HTTP/1.1
Host: [host]
Connection: keep-alive
Accept-Encoding: identity
Accept: */*
User-Agent: AnyConnect Windows 5.0.01242
X-Support-HTTP-Auth: false
X-Transcend-Version: 1
X-Aggregate-Auth: 1
X-Pad: 00000000000000000000000000000000000000000000
Content-Type: application/x-www-form-urlencoded
Content-Length: [content_length]

<?xml version="1.0" encoding="UTF-8"?>
                <config-auth client="vpn" type="init" aggregate-
auth-version="2">
                <version who="vpn">5.0.01242</version>
                <device-id computer-name="intel-PC" device-
type=[tasking_data] platform-version="6.1.7601 Service Pack 1"
unique-
id="1A4569E3936C5A234C512BBF1F2257A3D6D0D377308E1697001DD12510F
27E57">win</device-id>
                <mac-address-list>
                <mac-address public-interface="true">31-0c-19-
72-5b-a3</mac-address></mac-address-list>

                <opaque is-for="sg">
                <tunnel-group>DefaultWEBVPNGroup</tunnel-group>
                <group-alias>annyconnect</group-alias>
                <auth-handle>0</auth-handle>
                <config-hash>0</config-hash></opaque>

                <group-select>111</group-select>
                <group-access>https://333333/</group-access>

                </config-auth>
```

Appendix 3 – VPN client authentication tasking response

An example actor tasking response can be seen below.

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset=utf-8
Transfer-Encoding: chunked
Cache-Control: no-store
Pragma: no-cache
Connection: Keep-Alive
Date: [date]
X-Frame-Options: SAMEORIGIN
Strict-Transport-Security: max-age=31536000; includeSubDomains
X-Content-Type-Options: nosniff
X-XSS-Protection: 1
Content-Security-Policy: default-src 'self' 'unsafe-inline'
'unsafe-eval' data: blob:; frame-ancestors 'self'
X-Aggregate-Auth: 1

<?xml version="1.0" encoding="UTF-8"?>
<config-auth client="vpn" type="auth-request" aggregate-auth-
version="2">
<client-cert-request></client-cert-request>
<message>[response_data]</message>
</config-auth>
```

MITRE ATT&CK®

This report has been compiled with respect to the MITRE ATT&CK® framework, a globally accessible knowledge base of adversary tactics and techniques based on real-world observations.

Tactic	ID	Technique	Procedure
Persistence	T1542.003	Pre-OS Boot: Bootkit	RayInitiator persists by patching GRUB to hijack control of the boot process and injects malware into user-mode.
Defence Evasion	T1480.001	Execution Guardrails: Environmental Keying	LINE VIPER tasking payloads sent to victim devices are checked for multiple victim-specific tokens before they are run.
	T1014	Rootkit	LINE VIPER patches system integrity checks and returns results as if the device wasn't compromised. It also collects CLI commands which would enable monitoring of user activity.
	T1562	Impair Defenses	LINE VIPER supresses specific syslog messages to hide malware behaviour.
	T1562	Impair Defenses	LINE VIPER disables heap integrity verification functionality.
	T1202	Indirect Command Execution	LINE VIPER has the ability to run arbitrary CLI commands.
Credential Access	T1040	Network Sniffing	LINE VIPER has the capability to perform packet captures and has been observed collecting protocols with plaintext credentials.
Command and Control	T1071.001	Application Layer Protocol: Web Protocols	LINE VIPER is deployed and tasked via HTTPS WebVPN requests.

	T1095	Non-Application Layer Protocol	LINE VIPER can receive tasking via ICMP.
	T1571	Non-Standard Port	LINE VIPER responds to ICMP tasking via raw TCP using high-ephemeral ports.
	T1573.001	Encrypted Channel: Symmetric Cryptography	LINE VIPER receives tasking encrypted with a per-request AES key.
	T1573.002	Encrypted Channel: Asymmetric Cryptography	LINE VIPER uses an RSA public key to perform a symmetric key exchange.
Impact	T1529	System Shutdown/Reboot	LINE VIPER has an anti-forensic capability to immediately reboot the device when certain CLI commands are run. Additionally, LINE VIPER has a module to force a device reboot after a delay.

Disclaimer

This report draws on information derived from NCSC and industry sources. Any NCSC findings and recommendations made have not been provided with the intention of avoiding all risks and following the recommendations will not remove all such risk. Ownership of information risks remains with the relevant system owner at all times.

This information is exempt under the Freedom of Information Act 2000 (FOIA) and may be exempt under other UK information legislation.

Refer any FOIA queries to ncscinfoleg@ncsc.gov.uk.