Hunting Russian Intelligence “Snake” Malware

cisa.gov/news-events/cybersecurity-advisories/aa23-129a

Cybersecurity Advisory

Last Revised

May 09, 2023

Alert Code

AA23-129A

SUMMARY

The Snake implant is considered the most sophisticated cyber espionage tool designed and used by Center 16 of Russia’s Federal Security Service (FSB) for long-term intelligence collection on sensitive targets. To conduct operations using this tool, the FSB created a covert peer-to-peer (P2P) network of numerous Snake-infected computers worldwide. Many systems in this P2P network serve as relay nodes which route disguised operational traffic to and from Snake implants on the FSB’s ultimate targets. Snake’s custom communications protocols employ encryption and fragmentation for confidentiality and are designed to hamper detection and collection efforts.

We have identified Snake infrastructure in over 50 countries across North America, South America, Europe, Africa, Asia, and Australia, to include the United States and Russia itself. Although Snake uses infrastructure across all industries, its targeting is purposeful and tactical in nature. Globally, the FSB has used Snake to collect sensitive intelligence from high-priority targets, such as government networks, research facilities, and journalists. As one example, FSB actors used Snake to access and exfiltrate sensitive international relations documents, as well as other diplomatic communications, from a victim in a North Atlantic Treaty Organization (NATO) country. Within the United States, the FSB has victimized industries including education, small businesses, and media organizations, as well as critical infrastructure sectors including government facilities, financial services, critical manufacturing, and communications.

This Cybersecurity Advisory (CSA) provides background on Snake’s attribution to the FSB and detailed technical descriptions of the implant’s host architecture and network communications. This CSA also addresses a recent Snake variant that has not yet been widely disclosed. The technical information and mitigation recommendations in this CSA are provided to assist network defenders in detecting Snake and associated activity. For more

information on FSB and Russian state-sponsored cyber activity, please see the joint advisory Russian State-Sponsored and Criminal Cyber Threats to Critical Infrastructure and CISA's Russia Cyber Threat Overview and Advisories webpage.

Download the PDF version of this report:

Hunting Russian Intelligence "Snake" Malware (PDF, 4.11 MB)

INTRODUCTION

What is Snake?

We consider Snake to be the most sophisticated cyber espionage tool in the FSB's arsenal. The sophistication of Snake stems from three principal areas. First, Snake employs means to achieve a rare level of stealth in its host components and network communications. Second, Snake's internal technical architecture allows for easy incorporation of new or replacement components. This design also facilitates the development and interoperability of Snake instances running on different host operating systems. We have observed interoperable Snake implants for Windows, MacOS, and Linux operating systems. Lastly, Snake demonstrates careful software engineering design and implementation, with the implant containing surprisingly few bugs given its complexity.

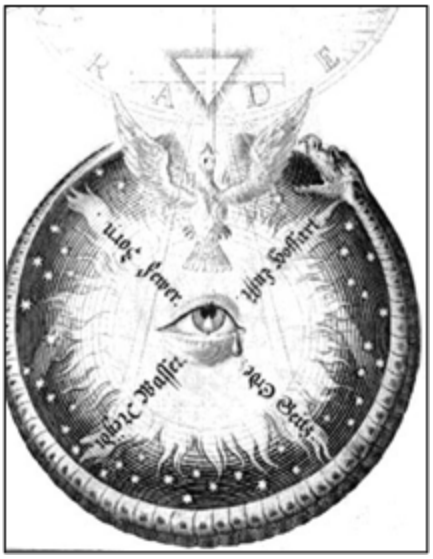
Following open source reporting by cybersecurity and threat intelligence companies on Snake tactics, techniques, and procedures (TTPs), the FSB implemented new techniques to evade detection. The modifications to the implant enhanced challenges in identifying and collecting Snake and related artifacts, directly hampering detection from both host- and network-based defensive tools.

The effectiveness of this type of cyber espionage implant depends entirely on its long-term stealth, since the objective of an extended espionage operation involves remaining on the target for months or years to provide consistent access to important intelligence. The uniquely sophisticated aspects of Snake represent significant effort by the FSB over many years to enable this type of covert access.

Background

The FSB began developing Snake as "Uroburos" in late 2003. Development of the initial versions of the implant appeared to be completed around early 2004, with cyber operations first conducted using the implant shortly thereafter. The name Uroburos is appropriate, as the FSB cycled it through nearly constant stages of upgrade and redevelopment, even after public disclosures, instead of abandoning it. The name appears throughout early versions of the code, and the FSB developers also left other unique strings, including "Ur0bUr()sGoTyOu#", which have publicly come back to haunt them.

Unique features in early versions of Uroburos included a low resolution image of a portion of a historical illustration of an uroboros by the German philosopher and theologian Jakob Böhme. One approach to a tertiary backdoor used this image as the key. The same image had also been embedded in other Snake-related components. The image, blown up to a higher resolution, is shown below.



In addition, early FSB developers of the Snake implant left portions of unique code throughout the implant which reveal inside jokes, personal interests, and taunts directed at security researchers. For instance, the “Ur0bUr()sGoTyOu#” string referenced above was replaced with “gLASs D1cK” in 2014 following some of the public cybersecurity reporting.

Attribution

We attribute Snake operations to a known unit within Center 16 of the FSB. This unit more broadly operates the numerous elements of the Turla toolset, and has subunits spread throughout Russia in a reflection of historical KGB signals intelligence operations in the Soviet Union. Snake has been a core component of this unit’s operations for almost as long as Center 16 has been part of the FSB. The extensive influence of Snake across the Turla toolset demonstrates its impact on practically every aspect of the unit’s modern era of cyber operations.

Daily operations using Snake have been carried out from an FSB facility in Ryazan, Russia, with an increase in Snake activity during FSB working hours in Ryazan, approximately 7:00 AM to 8:00 PM, Moscow Standard Time (GMT+3). The main developers were Ryazan-based FSB officers known by monikers included in the code of some versions of Snake. In addition to developing Snake, Ryazan-based FSB officers used it to conduct worldwide operations; these operations were different from others launched from Moscow or other FSB sites based on infrastructure and techniques.

While the development and re-tooling of Snake has historically been done by Ryazan-based FSB officers, Snake operations were also launched from an FSB Center 16-occupied building in Moscow. Our investigations have identified examples of FSB operators using Snake to its full potential, as well as FSB operators who appeared to be unfamiliar with Snake's more advanced capabilities. These observations serve to illustrate the difficulty in using such an advanced toolset across the various geographically dispersed teams comprising this unit within FSB Center 16.

We have been collectively investigating Snake and Snake-related tools for almost 20 years, as well as other operations by this unit since the 1990s. During that time, the FSB has used Snake in many different operations, and they have demonstrated the value placed in this tool by making numerous adjustments and revisions to keep it viable after repeated public disclosures and other mitigations. Snake's code and multiple Snake-related tools have been either a starting point or a key influence factor for a diverse range of other highly prolific implants and operational tools in the Turla family. Most notably, this has included Carbon (aka Cobra)—derived from Snake's code base—and the similarly Snake-adjacent implant Chinch (currently known in open sources as ComRAT).

Victimization

We have identified Snake infrastructure in over 50 countries across North America, South America, Europe, Africa, Asia, and Australia, to include the United States and Russia itself. Although Snake leverages infrastructure across all industries, its targeting is purposeful and tactical in nature. For instance, if an infected system did not respond to Snake communications, the FSB actors would strategically re-infect it within days. Globally, the FSB has used Snake to collect sensitive intelligence from high priority targets, such as government networks, research facilities, and journalists. As one example, FSB actors used Snake to access and exfiltrate sensitive international relations documents, as well as other diplomatic communications, from a victim in a NATO country. Within the United States, the FSB has victimized industries including education, small businesses, and media organizations, as well as critical infrastructure sectors including government facilities, financial services, critical manufacturing, and communications.

Other Tools and TTPs Employed with Snake

The FSB typically deploys Snake to external-facing infrastructure nodes on a network, and from there uses other tools and TTPs on the internal network to conduct additional exploitation operations. Upon gaining and cementing ingress into a target network, the FSB typically enumerates the network and works to obtain administrator credentials and access domain controllers. A wide array of mechanisms has been employed to gather user and administrator credentials in order to expand laterally across the network, to include keyloggers, network sniffers, and open source tools.

Typically, after FSB operators map out a network and obtain administrator credentials for various domains in the network, regular collection operations begin. In most instances with Snake, further heavyweight implants are not deployed, and they rely on credentials and lightweight remote-access tools internally within a network. FSB operators sometimes deploy a small remote reverse shell along with Snake to enable interactive operations. This triggerable reverse shell, which the FSB has used for around 20 years, can be used as a backup access vector, or to maintain a minimal presence in a network and avoid detection while moving laterally.

Snake Architecture

Snake's architectural design reflects professional software engineering practices. Critical pathways within the implant are made of stacks of loosely coupled components that implement well-designed interfaces. In addition to facilitating software development and debugging, this construction allows Snake to use multiple different components for the same purpose, choosing the specific component based on environmental considerations. For example, Snake's custom network communications protocols function as a stack. All implementations use an encryption layer and a transport layer, such as Snake's custom HTTP or raw TCP socket protocol. Each layer of the Snake network protocol stack solely implements a specified interface for operability with the two adjacent layers. The encryption layer and underlying transport layer thus function independently, so any custom Snake network protocol can employ an encryption overlay without any change to the encryption layer code.

This modularity allows Snake operators to choose the most logical network transport for the given environment without affecting Snake's other functionality. When using a compromised HTTP server as part of the Snake P2P network, the operators can ensure that all traffic to this machine follows the Snake custom HTTP protocol and thereby blends effectively with legitimate traffic. In the context of a compromised machine that legitimately allows secure shell (SSH) connections, Snake can utilize its custom raw TCP socket protocol instead of its custom HTTP protocol. All other layers of the Snake protocol stack, from the immediately adjacent transport encryption layer to the distant command processing layer, can and do remain entirely agnostic to the transport layer as long as it implements its interface correctly. This architecture also allows the Snake developers to easily substitute a new communications protocol when they believe one has been compromised, without necessitating any downstream changes in the code base. Lastly, this design facilitates the development of fully interoperable Snake implants running on different host operating systems.

Snake's technical sophistication extends from the software architecture into the lower-level software implementation. Original versions of Snake were developed as early as 2003, before many of the modern programming languages and frameworks that facilitate this type of modular development were available. Snake is written entirely in C, which provides

significant advantages in low-level control and efficiency, but which does not provide direct support for objects or interfaces at the language level and provides no assistance with memory management. The developers of Snake successfully implemented the implant's complex design in C with very few bugs, including careful avoidance of the common pitfalls associated with null-terminated strings and the mixing of signed and unsigned integers. Additionally, the developers demonstrate an understanding of computer science principles throughout the implant's implementation. This includes selecting and correctly coding asymptotically optimal algorithms, designing and utilizing efficient custom encoding methodologies that closely resemble common encoding schemes, and handling the numerous possible errors associated with systems-level programming in a secure manner.

Capitalizing on Mistakes

Although the Snake implant as a whole is a highly sophisticated espionage tool, it does not escape human error. A tool like Snake requires more familiarity and expertise to use correctly, and in several instances Snake operators neglected to use it as designed. Various mistakes in its development and operation provided us with a foothold into the inner workings of Snake and were key factors in the development of capabilities that have allowed for tracking Snake and the manipulation of its data.

The FSB used the OpenSSL library to handle its Diffie-Hellman key exchange. The Diffie-Hellman key-set created by Snake during the key exchange is too short to be secure. The FSB provided the function DH_generate_parameters with a prime length of only 128 bits, which is inadequate for asymmetric key systems. Also, in some instances of what appeared to be rushed deployments of Snake, the operators neglected to strip the Snake binary. This led to the discovery of numerous function names, cleartext strings, and developer comments as seen in the following figure.

Figure 1: Non-Stripped Function and Command Names

```
const:000000000016C840                dq 3EEh
const:000000000016C848                dq offset _name_get_mejirod_path
const:000000000016C850                align 20h
const:000000000016C860                dq 3EFh
const:000000000016C868                dq offset _name_get_queue_path
const:000000000016C870                align 20h
const:000000000016C880                dq 3F0h
const:000000000016C888                dq offset _name_get_libs_path
const:000000000016C890                align 20h
const:000000000016C8A0 qword_16C8A0   dq 64h                  ; DATA XREF: _snake_builtin_cmd+4↑o
const:000000000016C8A0                                        ; _snake_builtin_cmd+2C↑o
const:000000000016C8A8                dq offset _snake_cmd_id
const:000000000016C8B0                dq 65h
const:000000000016C8B8                dq offset _snake_cmd_ps
const:000000000016C8C0                dq 66h
const:000000000016C8C8                dq offset _snake_cmd_run
const:000000000016C8D0                dq 67h
const:000000000016C8D8                dq offset _snake_cmd_kill
const:000000000016C8E0                db 68h
```

SNAKE HOST-BASED TECHNICAL DETAILS

The FSB has quickly adapted Snake when its capabilities have been publicly disclosed by private industry. Snake therefore exists in several variants, as it has evolved over almost 20 years. This CSA focuses on one of the more recent variants of Snake that up until now has not been widely disclosed. Older variants of Snake will be discussed briefly where applicable, but not discussed in depth, as many details of earlier Snake variants already exist in the public domain.

Installer

The Snake installer has gone by various names throughout Snake's existence (e.g., "jpinst.exe"). This advisory will describe the version of the installer which regularly used the name "jpsetup.exe". This executable is packed using a customized obfuscation methodology. The developers appear to have added the unpacking functionality from an open source project for viewing JPEG files. This technique serves to obfuscate the unpacking code within an otherwise legitimate code base. The unpacking code extracts an executable, herein referred to as the "Png Exe", and it extracts an AES encrypted blob from the Png Exe's resources, which herein will be referred to as the "Png Resource".

The jpsetup.exe installer requires two arguments to be passed via the command line for execution. The first argument is a wide character string hashed with SHA-256 twice, and the resulting value of these computations becomes the AES key that decrypts the Png Resource. The AES initialization vector (IV) consists of the first 16 bytes of the second argument to jpsetup.exe after prepending the argument with a wide character "1" string. Once decrypted, the Png Resource becomes an executable that will be referred to herein as "Stage 2".

When unpacked, many components are extracted from Stage 2's resources. Several of the resources are executables with additional resources of their own. Stage 2 creates structures from its resources, which ultimately become the host artifacts of Snake.

On-Disk Components

As Windows has been the most prevalent operating system targeted by Snake, this document will only discuss the Windows-based artifacts; however, Snake can be cross-compiled and is capable of running on other operating systems.

On-Disk Obfuscation

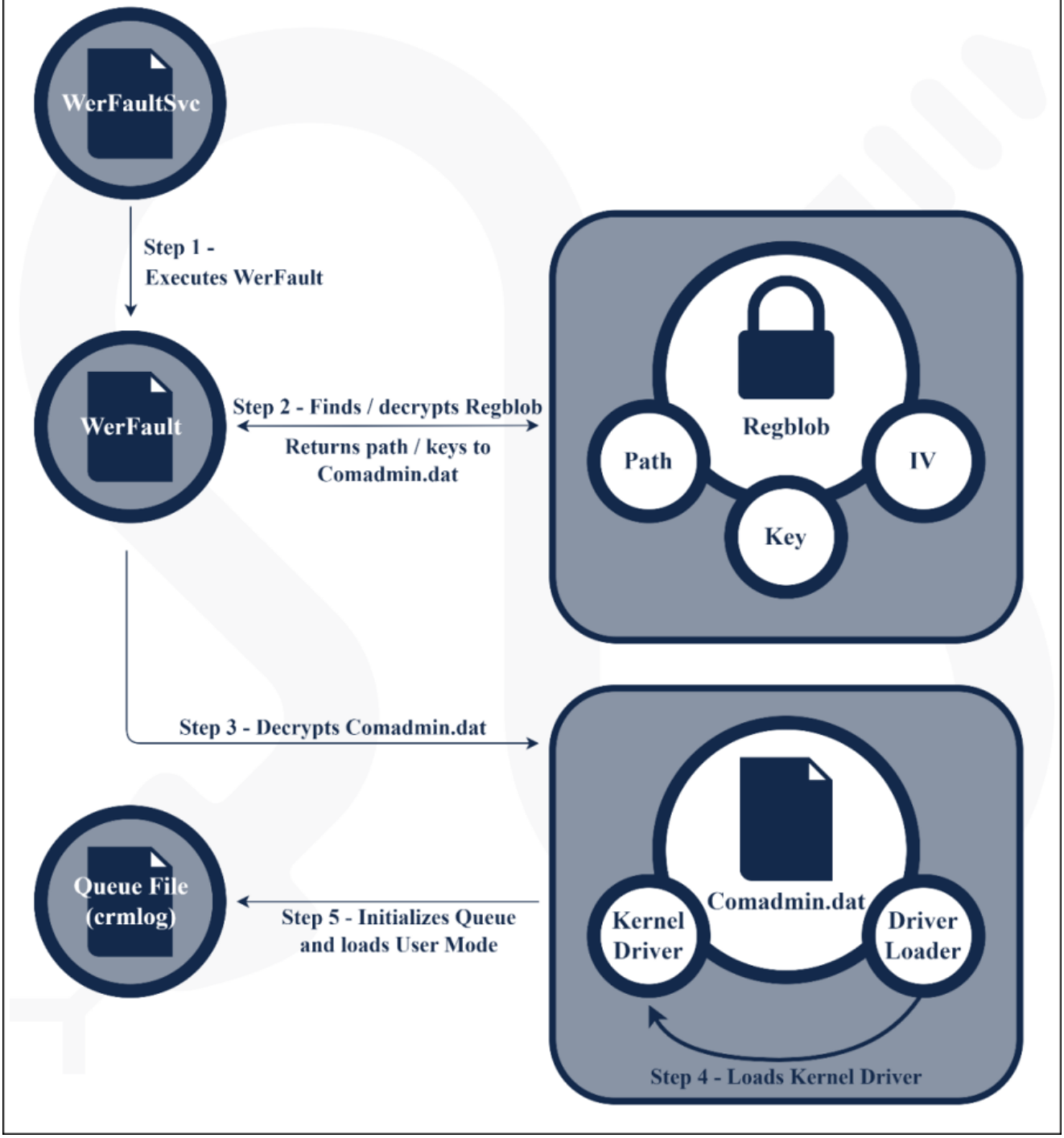
Snake's host architecture and network communications allow an unusual level of stealth. Snake makes inventive use of its kernel module in both of these contexts. All known Windows versions of Snake have used a concealed storage mechanism to hide host componentry. In addition to using the kernel module to remove the relevant components from any listing returned by the operating system, Snake utilizes the kernel module to mediate any

requests between Snake's user mode components and the concealed storage mechanism, which itself is encrypted with a unique per-implant key. This unique keying creates detection difficulties even for tools that are independent of the compromised operating system, since simple signatures targeting Snake host components would be ineffective.

Persistence Mechanism

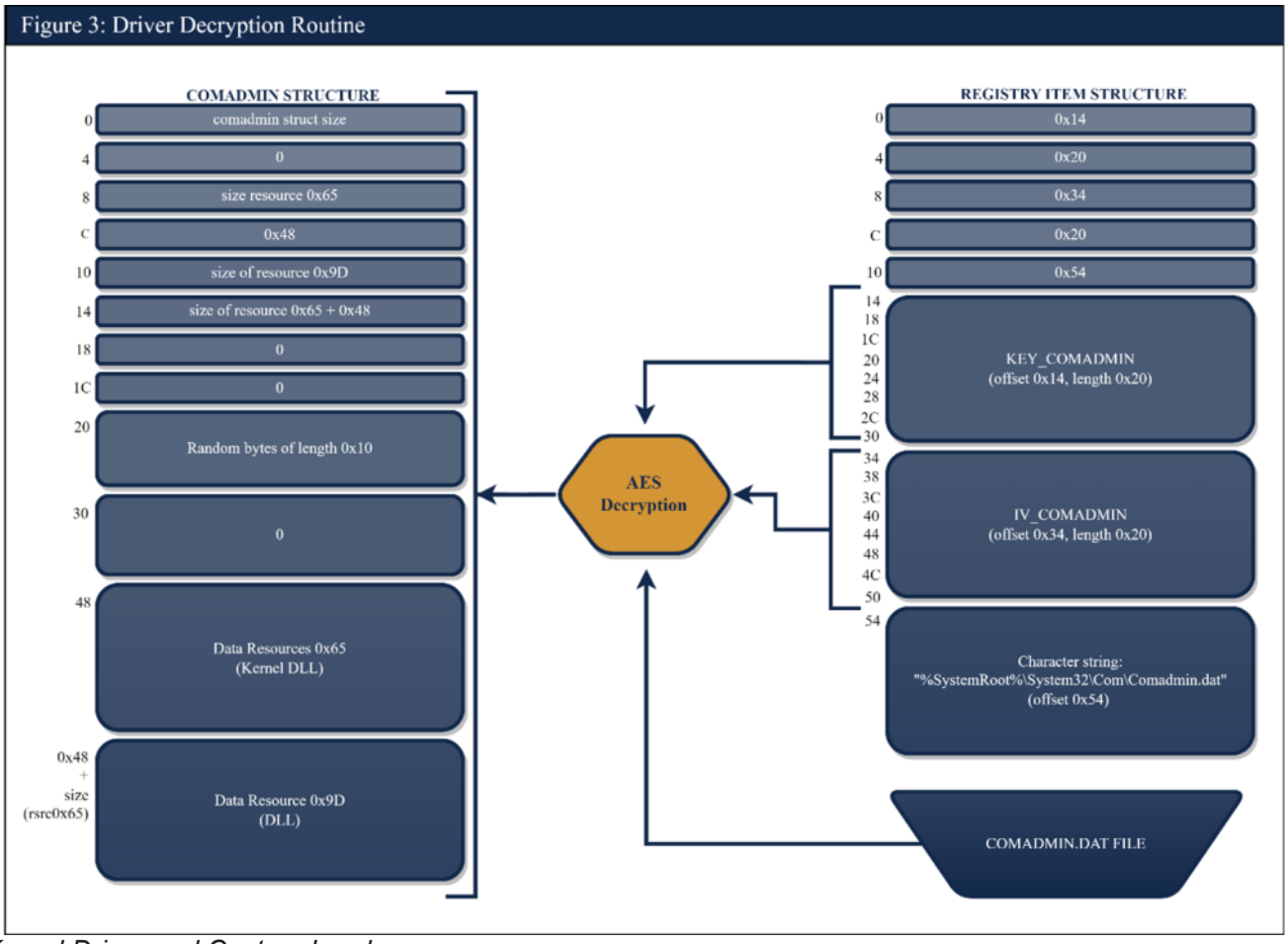
The Snake version primarily discussed in this advisory registers a service to maintain persistence on a system. Typically, this service is named "WerFaultSvc," which we assess was used to blend in with the legitimate Windows service WerSvc. On boot, this service will execute Snake's WerFault.exe, which Snake developers chose to hide among the numerous valid Windows "WerFault.exe" files in the %windows%\WinSxS\ directory. Executing WerFault.exe will start the process of decrypting Snake's components and loading them into memory.

Figure 2: Snake Boot Cycle



Encrypted Registry Key Data

Upon execution, Snake's WerFault.exe will attempt to decrypt an encrypted blob within the Windows registry that is typically found at HKLM:\SOFTWARE\Classes\.wav\OpenWithProgIds. The encrypted data includes the AES key, IV, and path that is used to find and decrypt the file containing Snake's kernel driver and kernel driver loader. The registry object's structure can be seen on the right side of the following figure. Snake uses Microsoft Windows Cryptography API: Next Generation (CNG) key store to store the AES key needed to decrypt the registry object.



Kernel Driver and Custom Loader

Snake's installer drops the kernel driver and a custom DLL which is used to load the driver into a single AES encrypted file on disk. Typically, this file is named "comadmin.dat" and is stored in the %windows%\system32\Com directory. The structure of this file can be seen on the left side of the figure above. The key, IV, and path to comadmin.dat are stored in the encrypted registry blob.

The Queue File

The last host-based artifact to discuss is the Queue File. Typically, this file has been found within the %windows%\Registration directory with the format of <RANDOM_GUID>.<RANDOM_GUID>.crmlog, and is decrypted by Snake's kernel driver. Due to the complexity and importance of the Queue File, its details are discussed at length in the following subsection.

The Queue

The Queue is a Snake structure that contains various pieces of information, including key material, communication channels, modes of operation, the principal user mode component, etc., that Snake requires for successful operation. It should be noted that this is a name used by the developers and is not equivalent to a "queue" in the normal context of computer science. The Queue data is saved on disk in the Queue File, which is a flat file with a

substructure that includes a 0x2c-byte file header followed by data blocks. Each data block corresponds to exactly one Queue Item, which could be, for example, a simple configuration parameter, a Snake command, or an entire embedded executable. Each Queue Item is associated with a specific Queue Container.

Queue Containers and Items

Each Container is identified by its Type and Instance values. Each Container Type holds the same type of information used by the Snake implant for a specific purpose. The following table shows the various Container Types and their functions. A Queue can have multiple Containers of the same Type, but each of these Containers will have different Instance values.

Table 1: Queue File Containers

CONTAINER NUMBER	CONTAINER USE
0x0	Incoming commands / data
0x1	Outbound commands / data
0x2	Mode configuration data
0x3	Embedded files / modules
0x4	Command logs
0x5	Network logs
0x6	Timed commands
0x7	Read agents_track data
0x8	Deprecated
0x9	Deprecated
0xb	Available modes of operation

The data in each Container in the Queue is separated into Queue Items with the 0x40-byte metadata structure shown in the following table. The data content of the Queue Item immediately follows this structure. The Queue Items in each Container are distinguished by their corresponding Item Number as well as their Item Type identifier. The Item Number is assigned by the Snake implant itself, while Snake operators generally refer to the Item Type value when trying to reference a specific item.

Table 2: Queue Item Structure

OFFSET	DESCRIPTION
0x0	Container Instance
0x4	Item Number
0x8	Container Type
0xC	Item Type
0x10	Parameter 0x10
0x14	Parameter 0x14
0x18	Parameter 0x18
0x1C	Parameter 0x1C
0x20	Create Time
0x24	Access Time
0x28	Modify Time
0x2C	Data Capacity
0x30	Data Length
0x34	Unknown
0x38	Unknown
0x3C	Unknown

Queue File Encryption

In previous versions of Snake, the Queue File existed within an encrypted covert store. The data belonging to the Queue Items themselves were also CAST-128 encrypted. In more recent versions, the covert store was removed, and the Queue File exists by itself on disk. The Queue Items inside the Queue File are still encrypted with CAST-128, and in addition, the full Queue File is also CAST-128 encrypted. The CAST keys used to encrypt the Queue

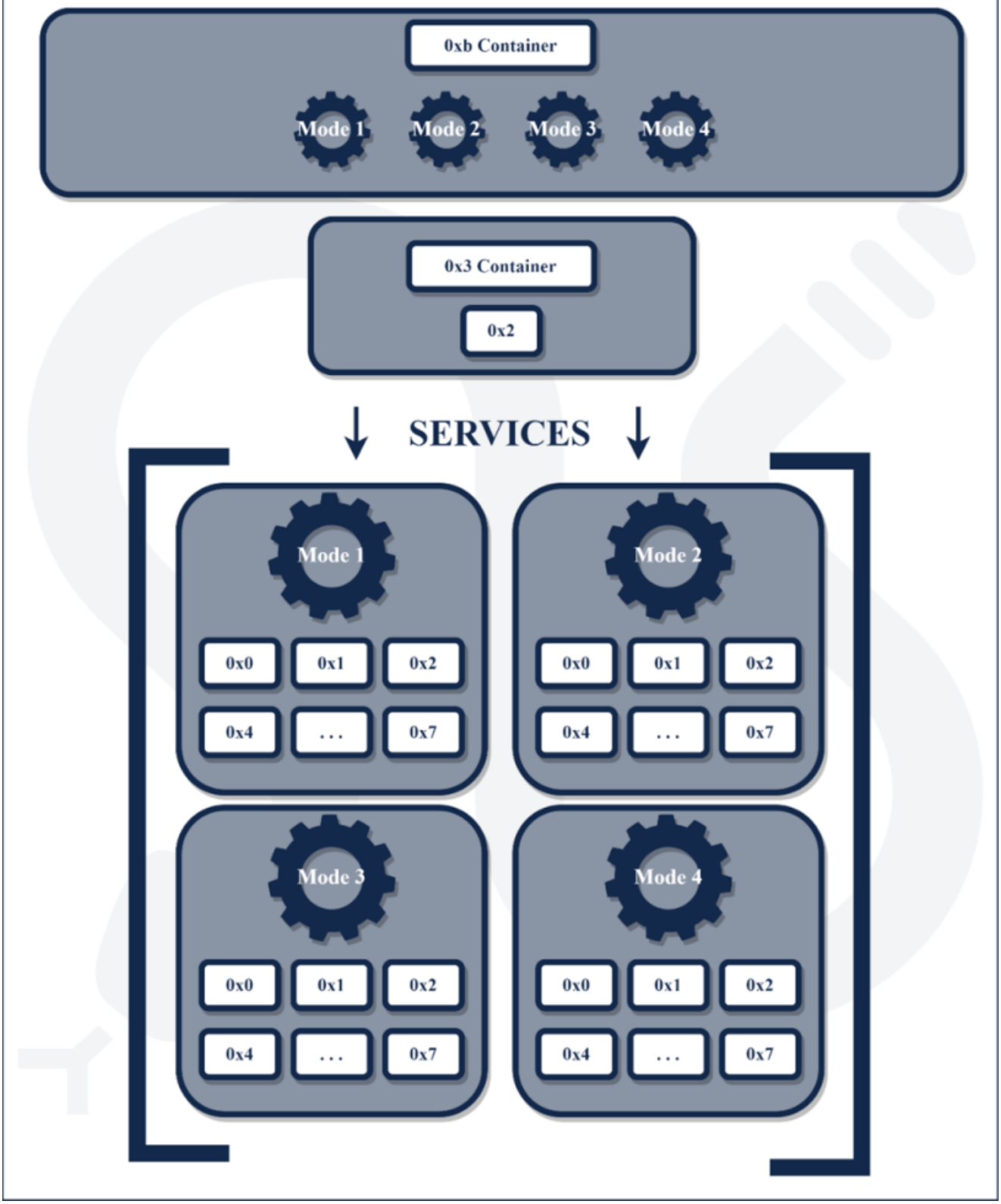
Items within a Container Instance can be found in that Instance's corresponding 0x2 Container as Item Type 0x229 (see below). The key and IV used to encrypt the Queue File can be found by decoding strings within Snake's kernel driver.

Container Descriptions

0xb Container

The 0xb Container lists the available modes of operation for a given Snake implant. When using a certain mode, Snake uses a specific set of Containers and communication channels. Each infection can use up to four different modes. Each mode in the 0xb Container will have a Container Instance value that all Containers associated with this mode will use, except for the 0x3 Container.

Figure 4: Queue File Container Organization



0x0 Container

The 0x0 Container handles incoming commands/data for the host of the Snake infection. Commands will be queued in this Container until the implant is ready to execute them.

0x1 Container

The 0x1 Container handles outbound commands/data for the host of the Snake infection. The data will be queued within the 0x1 Container until the implant is ready to exfiltrate them.

0x2 Container

The 0x2 Container holds the configuration information for the mode to which it corresponds. Various pieces of information vital to Snake's successful operation are stored within these Containers. This subsection will discuss a subset of the parameters that can be found within the 0x2 Container.

Table 3: Container 2 Queue Items

ITEM TYPE	DESCRIPTION
0x001	Queue instance value
0x00c	Host ustart value (only in Snake's kernel 0x2 Container)
0x064	Number of internal communication channels
0x065	Internal communication channel details 1
0x066	Internal communication channel details 2
⋮	⋮
0x06f	Internal communication channel details 10
0x070	Option parameters for communication channel with type 0x65
0x071	Option parameters for communication channel with type 0x66
⋮	⋮
0x07a	Option parameters for communication channel with type 0x6f
0x0c8	Number of IPs / domains available for passive use
0x0c9	IP / domain 1
0x0ca	IP / domain 2
⋮	⋮
0x0d3	IP / domain 10
0x0d4	Option parameters for IP / domain with type 0xc9
0x0d5	Option parameters for IP / domain with type 0xca
⋮	⋮
0x0de	Option parameters for IP / domain with type 0xd3
0x227	Public RSA key outbound communications
0x228	Public RSA key inbound communications
0x229	Queue item CAST encryption key for containers of matching instance value
0x28b	Process that was infected with Snake (only in Snake's kernel 0x2 Container)
0x2ff	Other potential injection locations (only in Snake's kernel 0x2 Container)

Pivotal key information can be found within the 0x2 Containers. This includes the inbound and outbound RSA keys (Items 0x228 and 0x227, respectively), the CAST key (Item 0x229) used to encrypt the individual items within the Queue Container, pre-shared keys used for the top layer of encryption in Snake's network communication protocol, and a quasi-unique value for the implant, called the "ustart" value, needed for Snake network connectivity.

Snake is constantly passing data between its kernel and user mode components. The methodology (generally, named pipes) used to make these communications is listed in Items 0x65-0x6f of the 0x2 Container. Items 0x70-0x7a list the parameters necessary to establish these communications.

Items 0xc9-0xd3 contain details of up to ten other Snake infections, referred to as "communication channels", which the implant can communicate with during Passive Operations. The parameters needed to establish Snake sessions with the other hosts can be found in Items 0xd4-0xde.

Many additional data points, such as the process name where Snake injected itself or the modules Snake has loaded from its 0x3 Container, can be found within 0x2 Containers.

0x3 Container

The 0x3 Container houses embedded files and modules for Snake. A single 0x3 Container will be accessible to all Containers in the Queue. The 0x3 Container has its own dedicated 0x2 Container that only includes a single Queue Item of Item Type 0x229 (a CAST-128 key). This key will be used to encrypt and decrypt all of the embedded files and modules within the 0x3 Container.

Table 4: Container 3 Queue Items

ITEM TYPE	DESCRIPTION
0x00001	Zlib library
0x00002	Retrieves logical drive info
0x00007	Custom Snake module
⋮	⋮
0x00018	Nettool portscanner
0x0001d	Dumpel.exe
0x00023	PsExec
⋮	⋮
0x04e21	JPEGView, installer
0xf4240	User mode (32 bit)
0xf4241	User mode (64 bit)
⋮	⋮

The Item Types assigned to the embedded files and modules within the 0x3 Container are consistent across all of the Snake infections within Snake's P2P network. For example, the 0x01 Item Type is the Zlib library, and therefore any time an Item Type of 0x01 is seen within the 0x3 Container of a Snake infection, that file is always the Zlib library. The implant's 0x2 Container will keep track of libraries that it has loaded. If the DLL is a file on disk, the full path to the DLL is saved in the 0x2 Container. If the library was loaded from a 0x3 Container, the loaded module will be displayed in the implant's 0x2 Container in the format "&<Item Type (within 3 container) of loaded module>".

0x4 Container

The 0x4 Container logs command activity. Each Queue Item within the Container is a log of a single executed or attempted command. Each mode will have its own corresponding 0x4 Container.

0x5 Container

The 0x5 Container holds Snake network logs, noting any IP address that has connected to this implant. Some versions of Snake no longer make use of this Container.

0x6 Container

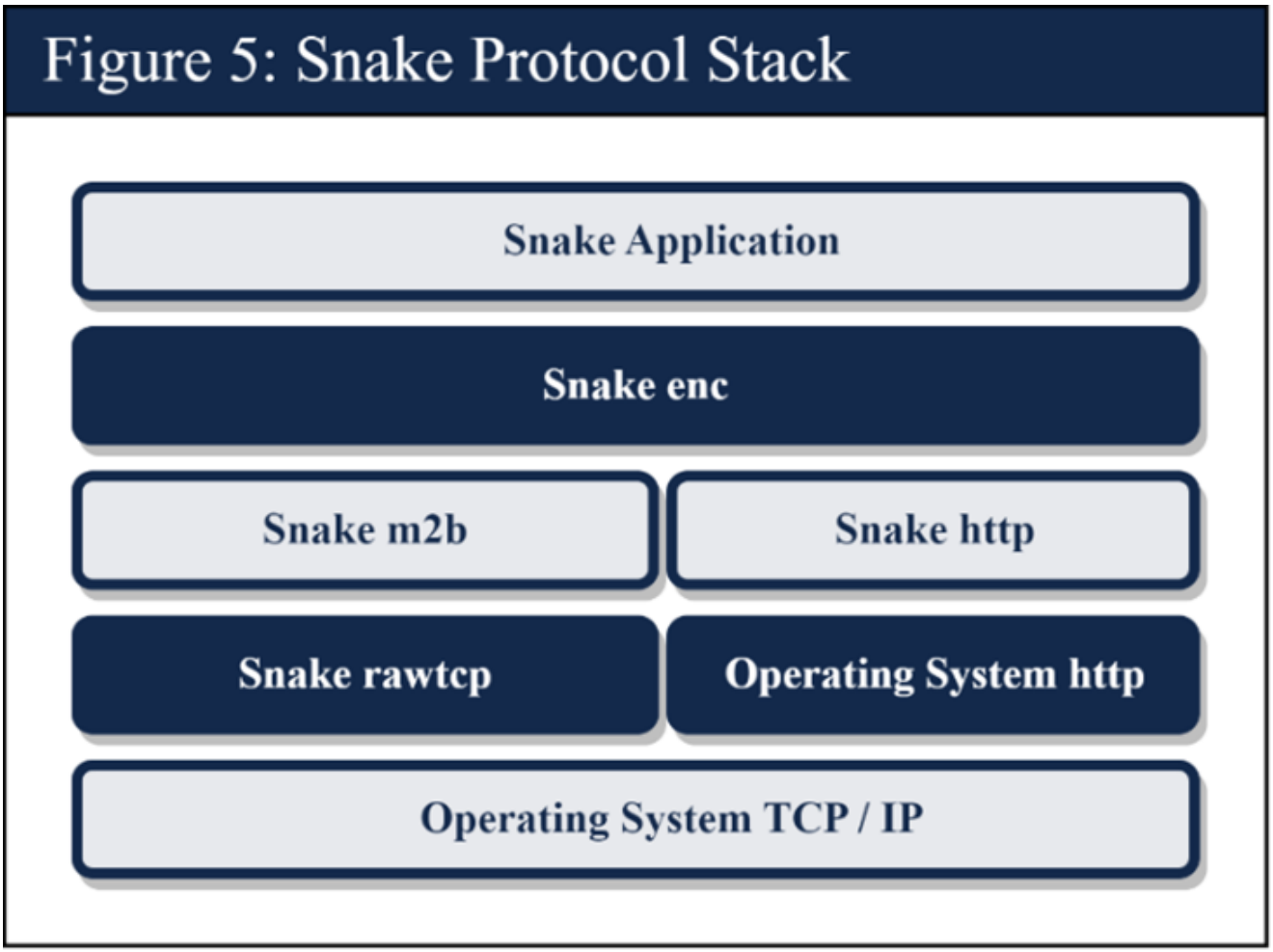
The 0x6 Container saves commands that are set to execute at specific times. A Queue Item is created for each scheduled command.

0x7 Container

The 0x7 Container logs the IP addresses of any other Snake implants that have connected to this implant during Passive Operations. The commands 0x79 (Read Agents Track) and 0x7a (Clear Agents Track) are used to interact with this Container. Note that the command 0x7a had been deprecated in some versions of Snake and returns the error “function unsupported” if called.

SNAKE NETWORK COMMUNICATIONS

Snake’s network communications are encrypted, fragmented, and sent using custom methodologies that ride over common network protocols, including both raw TCP and UDP sockets and higher-level protocols like HTTP, SMTP, and DNS. Snake’s protocols for HTTP and TCP are the most commonly seen, but functionality exists for UDP, ICMP, and raw IP traffic. Snake’s network communications are comprised of “sessions”, which are distinct from the sessions associated with the legitimate protocol it is riding on top of (e.g., TCP sessions). The Snake session is then comprised of distinct commands. Both Snake’s custom transport encryption layer (“enc”) and Snake’s Application Layer have their own encryption mechanisms, where the enc layer operates on an individual P2P session and the Snake Application Layer provides end-to-end encryption between the controller (i.e., point of origin) and the command’s ultimate destination. The following figure details Snake’s communication protocol stack.



Network Obfuscation

Snake's use of its kernel module also facilitates stealthy network communications. To participate fully in Snake's P2P network, implanted machines which are not the ultimate target must act as servers for other Snake nodes. Snake's kernel module, along with a thoughtfully designed mechanism for distinguishing Snake traffic from legitimate client traffic, allows the implant to function as a server in the Snake P2P network without opening any new ports, greatly complicating detection efforts. Additionally, Snake's custom network communication protocols are designed to blend with traffic that the compromised server normally would receive. This allows Snake operators to use legitimate servers as infrastructure, which reduces the effectiveness of simple IP address or domain blocking without needing to open new ports or send unusual looking traffic to this infrastructure.

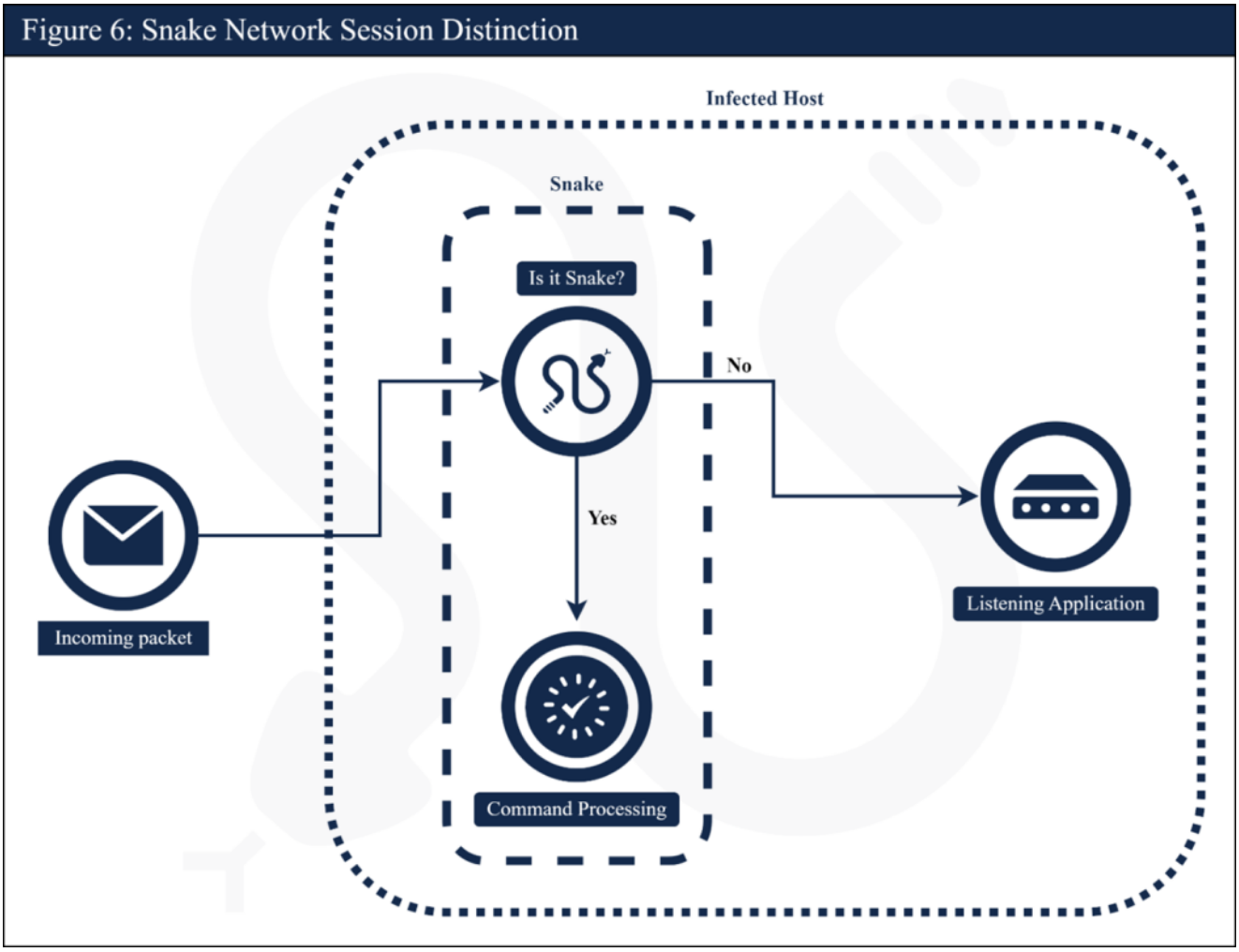
Snake's Network Authentication Technique ("ustart")

Snake uses its custom HTTP and raw socket TCP based protocols for large data communications. With these protocols and others, Snake employs a specific authentication mechanism to distinguish Snake traffic from legitimate traffic destined for application software on the compromised server. This technique enables one of the uniquely sophisticated aspects of Snake, which is its ability to function effectively as server software

without opening any further ports on the compromised system. The relevant per-implant authentication value is referred to as the “ustart” and is stored in the implant’s Queue File. There are multiple forms of the ustart value, including “ustart”, “ustart2”, and “ustartl”.

Rather than open a listening socket on a specified TCP port, the Snake kernel module intercepts the first client-to-server packet following the 3-way handshake in every TCP session. The kernel module then determines whether or not the contents of that packet are in fact valid for the ustart value of that target Snake implant. If so, the Snake kernel module forwards that packet and any future packets in the same TCP session to Snake’s own processing functionality, and the (presumably legitimate) application listening on that port remains unaware of this TCP session. If not, the Snake kernel module allows the packet—and the rest of the TCP session as it occurs—to reach the legitimate listening application, for example web server software. See the following for an illustration.

Figure 6: Snake Network Session Distinction



All of the ustart versions perform authentication by sending a random nonce along with data that comprises a mathematical operation on the combination of the nonce and the ustart value itself. The receiving machine then extracts the nonce and performs the same computations to authenticate the sending machine. The ustart2 and ustartl versions use the

Fowler-Noll-Vo (FNV) hash algorithm to generate the overall authentication value from the nonce and the ustart. This mechanism is slightly different in the custom Snake HTTP protocol versus the custom Snake TCP protocol.

Using the ustart methodology, a node in the Snake P2P network can function as a server without opening any otherwise closed ports and without interfering in the compromised server's legitimate functionality. Snake will only communicate over TCP ports on which another application is actively listening. This technique makes detecting Snake compromises through network traffic monitoring far more difficult. Inbound traffic to an unexpected TCP port can be detected or blocked using standard firewall or network intrusion detection functionality. Replacing a legitimate service application with a modified executable can lead to detection at either the host or network level. Snake's technique bypasses both of these mitigations. When combined with the fact that Snake traffic looks similar to expected traffic, especially in the case of Snake's HTTP based protocols, this renders detecting Snake communications difficult absent detailed knowledge of Snake's custom protocols.

Snake UDP

Outbound Communications via DNS Query

Snake uses a specialized communications protocol to encode information in seemingly standard DNS queries run via the Windows or POSIX API function gethostbyname, depending on the version.
Snake outbound DNS requests consist of character strings that are constructed to resemble standard domain names. The actual information being transmitted from the implant is contained in the part of the character string prior to the first '.' character. For illustration purposes, this subsection will outline how an arbitrary string of bytes is manipulated and then encoded to form an outbound Snake DNS request carrying data provided by the implant.

Snake outbound DNS requests originally take the form of byte arrays stored on the stack as the implant progresses through the communications function. The byte array has the following structure.

Table 5: DNS Byte Array

BYTE (BY OFFSET)	DESCRIPTION
0	Random byte, per communication
1	Random byte, per communication
2	Checksum byte
3	Random byte, per de-facto session
4	Flags byte
5	Data to be encoded

Only the low-order seven bits of the flags byte are used, and they have the following significance.

Table 6: Flags Byte

BITS	DESCRIPTION
0..4, inclusive	Length of data
5	Set iff the data is legitimate (non-random)
6	Set iff the implant will store the response

After calculating and obfuscating the byte array values shown above, Snake encodes these byte values as de-facto base32 text, using the ten digits 0-9 and the 26 lowercase ASCII letters a-z, with v, w, x, y, and z all corresponding to the same value, as only 32 distinct characters are needed. Snake then inserts '-' characters at specified locations and sends the DNS request using the gethostbyname function. The resulting encoded string mimics a legitimate DNS request; because characters after the first '.' are not part of the implant's communications, any arbitrary suffix (e.g., ".com") can be used.

Inbound Communications via DNS Query Response

After sending the encoded DNS request, Snake parses the returned information. In a normal DNS request, the returned hostent structure contains a list of IPv4 addresses as 32-bit unsigned integers if the domain resolves to one or more IPv4 addresses. In the Snake DNS protocol, these 32-bit integers represent the covert channel data. The Snake implant sorts

the 32-bit integers by the highest order nibble and then interprets the remaining 28 bits of each integer as the actual encoded data. The Snake DNS protocol thus provides a well-concealed, low-bandwidth communications channel. For larger bandwidth communications, Snake uses its custom HTTP and TCP protocols.

Snake HTTP

The most common custom protocol that Snake uses is its “http” protocol, which rides on top of standard HTTP. It generally looks like normal HTTP communications, including a lot of base64-looking strings, thus blending well with normal network traffic. There have been multiple iterations of Snake’s http protocol, though the differences are only in the encoding; once that is peeled away, the underlying Snake http protocol is the same. For the purposes of this document, Snake’s former version of HTTP will be referred to as “http” and its more recent version as “http2”.

Snake communications using http2 are contained within seemingly legitimate Application Layer HTTP communications. In the client-to-server direction, the implant data is contained within an HTTP header field of a GET request, unless the data is over a certain size (usually 256 bytes, but configurable). Observed field keys have included: Auth-Data, Cache-Auth, Cookie, and Cockie (note misspelling). This list is not exhaustive; any standard HTTP header field can be used. The communication itself is contained in the legitimate HTTP header field’s value, meaning the content following the ‘:’ character and any whitespace immediately thereafter. In HTTP GET requests, the implant generally uses the default path ‘/’, but this is not required and is configurable. Larger client-to-server Snake http2 requests are contained in the body of an HTTP POST request, and server-to-client communications are contained in the body of the HTTP response.

All client-to-server Snake http and http2 requests begin with the ustart authentication. The specifics vary with each ustart version, but in each case the random nonce and the computed function of the nonce and ustart value are encoded in a manner which closely resembles the rest of the Snake communication. Since Snake http and http2 implant sessions can span multiple TCP sessions, the ustart authentication mechanism is included in every client-to-server communication.

Base62 Encoding

Snake’s http2 protocol uses a custom base62 encoding scheme that has the following differences from base64. Base62 uses 62 semantically significant characters instead of 64. The ratio of encoded-to-decoded characters in base62 is less dense (11:8) than the ratio base64 can achieve (12:9). Also, base62 uses extraneous characters in certain instances that have no semantic significance.

The base62 characters of semantic significance are the 62 strict alphanumeric characters: [0-9A-Za-z]. The extraneous characters that can be present in a base62 string—but which have no semantic significance—are: '/', ';', '=', and '_' (underscore). When present, these characters are removed prior to performing the decoding process. A valid base62 string can have up to 11 of these extraneous characters. A regular expression for base62 is included in the Mitigations section of this CSA.

http and http2 Metadata Structure

After the base62 decoding is completed, if necessary, the remaining data begins with an 8-byte metadata structure that provides rudimentary sessionization on top of the stateless HTTP. Snake's http and http2 client-to-server communications have three de-facto parts, which are concatenated into a single HTTP header value. These parts are: 1) an announce or authentication string, 2) a custom metadata structure, and 3) payload data. The metadata structure consists of the following:

```
struct http_meta {
        uint32_t session_number;
        uint16_t communication_number;
uint8_t flags;
uint8_t checksum;
};
```

Snake uses the session_number and communication_number fields to provide its own custom sessionization on top of the stateless Application Layer HTTP protocol. The checksum byte serves to validate the integrity of the structure and must equal the sum of the first seven bytes modulo 256.

Snake TCP

Snake has the ability to communicate through POSIX-style TCP sockets. The implant's custom TCP protocol, which herein will be called "tcp", uses thereliability features of the underlying TCP protocol. Thus, in the implant's custom tcp protocol, the concept of a TCP session and an implant "session" are the same, whereas in the implant's custom http protocols, one implant session could span multiple Transport Layer TCP sessions. Since the implant's overall communications protocol is based on the idea of commands and responses, Snake depends on being able to specify the length of any given command and response so the recipient Snake node knows when a particular communication ends. Snake achieves this in the custom tcp protocol by prefacing eachcommunication with its length encoded as a 32-bit big-endian unsigned integer.

Immediately following the TCP 3-way handshake, the implant completes the ustart authentication for this session. Since Snake tcp sessions are mapped one-to-one with an underlying protocol TCP session, the ustart authentication only occurs once per session, rather than with each client-to-server communication as in Snake http and http2. The Snake

tcp ustart mechanism is similar to the Snake http and http2 mechanisms, except that for certain ustart versions, Snake tcp uses a raw binary ustart which is not encoded in printable characters.

After the ustart authentication, the implant will begin sending length-data pairs. These pairs can be sent in the same packet or in two (or theoretically more) separate packets, but the pattern of length-data pairs will be present in each half of the stream (i.e., each direction) for the entirety of the implant communications for the remainder of the TCP session. Specifically, a length-data pair will consist of the length encoded as a big-endian 32-bit unsigned integer followed by data of exactly that length. For example, consider the instance where the implant is sending the following 4 arbitrary bytes:

```
89 ab cd ef
```

The on-wire communication from the implant would send the integer value 4 encoded as a big-endian 32-bit integer, followed by the actual 4 bytes themselves, as shown below. This could be split across two (or theoretically more) packets.

```
00 00 00 04 89 ab cd ef
```

The custom tcp protocol (as well as all custom http protocols) have been used in conjunction with the Snake enc protocol. Details of the Snake enc protocol are provided in the following subsection. Due to the manner in which the Snake enc and Snake tcp protocols interact, the first six length-data pairs of each TCP half-stream (following the single client-to-server announce or authentication packet described above) will have known lengths. Specifically, each half-stream will begin with length-data pairs of the following lengths: 0x8, 0x4, 0x10, 0x1, 0x10, 0x10. Note that these are the lengths of the raw data, so each communication will be preceded by a 4-byte big-endian integer specifying the corresponding length. Thus, one of the half-streams could have the following TCP content:

```
00 00 00 08 12 34 56 78 9a bc de f0
00 00 00 04 89 ab cd ef
00 00 00 10 12 34 56 78 9a bc de f0 12 34 56 78 9a bc de f0
00 00 00 01 12
00 00 00 10 12 34 56 78 9a bc de f0 12 34 56 78 9a bc de f0
00 00 00 10 12 34 56 78 9a bc de f0 12 34 56 78 9a bc de f0
```

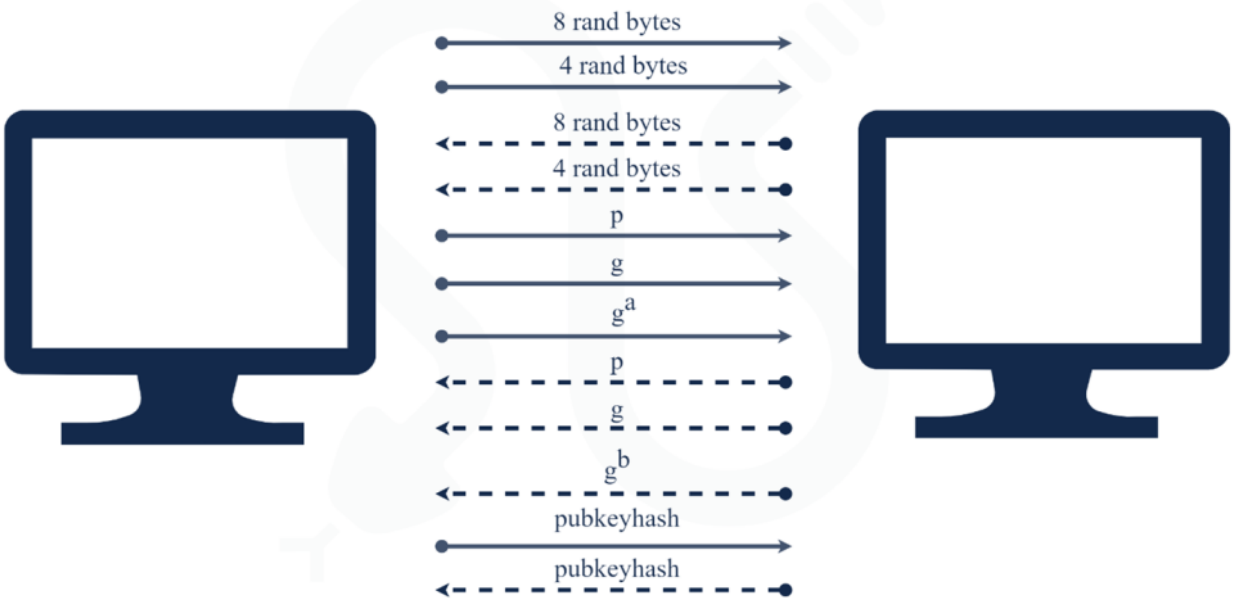
Snake “enc” Layer

As described above, Snake communications are all comprised of “Snake sessions”, irrespective of whichever legitimate protocol Snake is operating on top of. Snake’s top layer of encryption, called the enc layer, utilizes a multi-step process to establish a unique session key. The session key is formed through the combination of a Diffie-Hellman key exchange mixed with a pre-shared key (PSK) known to both parties. This PSK is stored in one of the communication channels, stored within the Queue.

The overall establishment of the session key requires 12 communication steps, six in each direction, which involve sharing the pseudo-random values used in the Diffie-Hellman exchange process as well as custom aspects of the Snake session key derivation method. The session key is used to encrypt the command headers and (inner) encrypted payloads.

This is the layer in which the critical error of providing a value of 128 bits instead of 128 bytes for the call to DH_generate_parameters within the OpenSSL library occurred. Due to this insufficient key length, breaking the Diffie-Hellman portion of the exchange is possible. Note that in the following figure, the variables ‘p’, ‘g’, ‘a’, and ‘b’ are used in standard descriptions of Diffie-Hellman.

Figure 7: Snake Session Key Establishment



SNAKE APPLICATION LAYER

Snake’s Application Layer is used to process Snake commands. The payload data for a Snake session can contain one or more command exchanges, which include both the incoming data sent to the implant as well as the response returned to the server. Each command is associated with a specific ordinal, and due to Snake’s modular design, operators are able to add new commands to extend Snake’s capabilities by remotely loading a new module.

The Snake implant differentiates between High and Low commands and handles them differently, based on the ordinal number range. The majority of Snake commands are High commands that have an ordinal of 0x64 (100 decimal) or higher. There are far fewer Low commands, and these include the Forwarding command (with ordinal 0x1), and the four

Queue commands (with ordinals 0xa, 0xb, 0xc, and 0xd). While Low commands are mostly used for moving data across the network, the High commands give the operator many options for interacting with an infected system.

Command 0x15-byte Header

All commands begin with a 0x15-byte header, followed by optional command parameter data; only some commands require parameters for successful execution. For example, the command Get, which exfiltrates a file, requires the name of the file to exfiltrate, whereas the command Process List, which returns a process listing, does not require any parameters.

The most important Command Header field contains the integer ordinal of the command being sent. The Item UID field represents a unique identifier for each individual command instance, and these values increase sequentially. The header has two fields used when a command is set to run at a specified date and time; these commands will be written to the 0x6 Container.

Some Low commands have another header before the payload data, which will be detailed below. All other commands have only the Command Header followed by the encrypted parameter data.

Figure 8: 0x15 Command Header

Command Encryption

Underneath Snake http2 or tcp encryption at the session layer, each command exchange is further encrypted. In older versions of Snake, the exchanges were CAST-128 encrypted using a different key for incoming and outgoing data. These keys were saved in the 0x2 Container in the 0x227 and 0x228 Items. The incoming payload data, if parameter data was present, could be decrypted with the 0x227 CAST key. Any response data was encrypted with the 0x228 CAST key.

In recent versions, the 0x227 and 0x228 Items hold two RSA-4096 public keys. For each side of an exchange, a new 16-byte CAST key is created with Microsoft's CryptoAPI CryptGenRandom function to obtain 16 random bytes. This key is used to CAST-128 encrypt the parameter or response data.

For an incoming command, the CAST key is signed (not encrypted) by the private key corresponding to the public key on the node to create a 512-byte RSA data blob. The incoming payload has the RSA blob, followed by the optional parameter data, which is CAST-128 encrypted. Snake uses the 0x227 RSA public key to decrypt the RSA blob, recover the CAST key, then decrypt the parameter data.

For an outgoing command, a new CAST key is obtained from CryptGenRandom, and any response data is CAST-128 encrypted. The key is then encrypted using the 0x228 public key to create a 512-byte data blob. The response payload data contains the 512-byte RSA blob, followed by the encrypted response data, when present.

Command Decoding

The implant will expect data in a specific format for each command ordinal. Parameter and response data contain several possible underlying data types, including wide-character plaintext strings, numeric values, data tables, files, or a combination of multiple types.

The parameter data buffer itself will be formatted in a specific way, depending on the command ordinal. Some commands have required parameters, as well as optional parameters. Commands with optional parameters will include a metadata header with the data length and data type (e.g., bool, integer, text, or data buffer) before the optional parameter's data. Other commands will expect the parameters to be formatted with length-data pairs, consisting of the parameter data length encoded as a four-byte big-endian integer followed by data of exactly that length. Still other commands have a custom header or will expect no length or metadata and will simply send the parameter data alone.

The response data will similarly be formatted by the implant in a specific way according to the command ordinal. The response data typically does not have a length or metadata preceding it, with the exception of the data tables. Examples of commands that return a table are the Process List command and the List Dir command.

Response data that includes a table will start with a table description header that indicates the number of columns and rows in the table. In addition, the header will include a Column Descriptor structure to indicate the type of data that column will contain, for example a string, uint32 or uint64, timestamp in epoch format, or the contents of a whole file (included as a table entry).

After the table description header, each field is added to the data payload buffer one at a time in a length-data pair. The fields across the first row are added in order, then the fields across the second row are added immediately after the first row with no metadata or separation, and so on. To parse this table, the server will account for the number of columns to determine where the next row starts.

High Commands

High commands are those with an ordinal of 0x64 (decimal 100) or higher. High commands give the operator many options for interacting with an infected system, as well as implant components. This subsection will describe some examples of the many High commands that can exist in the implant.

Some of the most basic High commands will gather information about the machine and return the results. For example, the FSB operators can use the PS command (0x65) to return a list of running processes, the List Dir command (0x840) to list the contents of a directory, or the Syst command (0x6b) to gather basic system information.

There are several commands that interact with the infected machine using standard built-in OS tools. The operator can use the Kill command (0x67) to kill a process, the Get command (0x68) to exfiltrate a file, the Put command (0x69) to write a file, the Del command (0x6a) to delete a file, or the Run command (0x66) to execute a command in a terminal shell and receive the results. For example, operators have used the Run command to run PowerShell commands, ping other hosts, use the Windows “net use” command to map network drives, and to run executable files that had been previously written to the node using the Put command.

Table 7: Examples of Snake Command Ordinals

COMMAND ORDINAL	DESCRIPTION
0x1	Snake forward
0xa	Snake queue enumerate
0xb	Snake queue read
0xc	Snake queue write
⋮	⋮
0x64	Snake ID
0x65	Snake process list
0x66	Snake run
⋮	⋮
0x70	Snake get config
0x71	Snake set config value
0x72	Snake modules load
0x73	Snake modules unload
0x74	Snake stop
⋮	⋮
0x840	Snake directory walk
0x844	Snake discovered hosts
⋮	⋮

In addition to commands that use the built-in OS functionality, there are several High commands that interact with Snake components. An operator can use the Read Config command (0x70) to read the 0x2 Container, which contains configuration data, or the Set Config Item command (0x71) to set a specific Queue Item within the 0x2 Container. For example, operators have used the Set Config Item command to add or update the IP addresses or domains and option parameters used to communicate with other Snake nodes. The Read Agents Track and Clear Agents Track commands (0x79 and 0x7a) interact with the 0x7 Container to read or delete logs which track which other Snake nodes have connected to this node. Note that the 0x7a command has been deprecated in some versions of Snake and returns the error “function unsupported” if called.

Snake has the ability to add additional commands by loading new modules. New modules can be loaded using the Load Modules command (0x72) or directly into memory using the Load Modules Mem command (0x7f). When compiling a module, the developer will assign an ordinal to each constituent command, which will then be used by the operator to call the newly added commands. These loaded modules can be removed using the Module Unload command (0x73).

Queue Commands

Queue Command Header

The four Queue commands contain a 0x3d-byte Queue Header following the Command Header. In more recent versions of Snake, this header is encrypted using the same CAST key used to encrypt the payload data. In this case, the Command Header is followed by the 512-byte RSA encrypted CAST key blob, the encrypted Queue Header, and finally the encrypted payload data.

Figure 9: Queue Command Header

Even though each of the four Queue commands only use a subset of the fields of the Queue Header (in different ways), the full header must be present for the command to be considered valid by the implant. Two fields in the header that all four Queue commands use are the Container Instance and Container Type fields, which indicate the specific Container on a node the Queue command intends to interact with. In the Queue Read and Write commands, the Item Type field is used to track the specific commands and their responses in the Containers.

Queue Enumerate Command

The Queue Enumerate command, with ordinal 0xa, is used to enumerate the contents of the 0x0 or 0x1 Containers to list all incoming or outgoing commands, respectively. The enumeration returns the 0x40-byte structure described above for each Queue Item, concatenated into a single return buffer.

Queue Read Command

The Queue Read command, with ordinal 0xb, is used to read an Item from the specified 0x0 or 0x1 Container. Several relevant fields in the Queue Header determine how the data is sent and stored. For example, the header determines whether the data should be sent immediately back to the server or stored for later transport. The header indicates if the implant should send the Queue Item's header (i.e., the same 0x40-byte metadata structure

returned by the 0xa command), the Item's data, or both. The header also indicates whether the Queue Item should be deleted after being read and can also indicate that Queue Items with a lower Item Type should be deleted. This allows FSB operators to clear out all command Items previous to the one being read.

Queue Write Command

The Queue Write command, with ordinal 0xc, is used to write a Queue Item to the specified 0x0 or 0x1 Container. The Queue Header will indicate if a new Queue Item will be created, or an existing Queue Item will be modified.

If a Queue Item is set to be modified, an Item with the specified Item Type must exist in the specified Container. Several fields in the header must match specific attributes of the existing Queue Item. If these checks are met, the parameter data is written to the Queue Item. Fields in the Queue Header will indicate the length of data to be written, and the offset into the existing Queue Item where the write should begin.
If a Queue Item is set to be created, Snake will delete existing Queue Items of the specified Item Type in the Container of interest, then create a new Item of the specified Item Type and write the parameter data to the Queue Item. A field in the Queue Header will indicate the length of data to be written.

Queue Delete Command

The Queue Delete command, with ordinal 0xd, is used to delete a Queue Item from the specified 0x0 or 0x1 Container. The Flags field will determine if the single Queue Item should be deleted, or if all Queue Items with a lower Item Type should be deleted as well.

Forward Commands

Forward commands, with command ordinal 0x1, are used to tell an implant to forward a Snake command to a second target node, where the command will be executed. The target node sends the response data back to the first implant, which will then package that response data as its own response back to the caller.

The command is designed to tell an implant to forward one command to another implant, but in practice, Forward commands are often built on top of each other to create a chain of hop points that will continue to forward a command to an end point, where it will be executed. The response data is then sent back through the same chain of hop points until it reaches the operator.

The Forward command has a 0x199-byte Forward Header, followed by the encrypted command parameter data that will be sent to the target node of the Forward command. The Forward Header contains the information the implant will need to connect to the target node, including the ordinal of the Snake command that is being forwarded to the target node for execution.

The implant that receives the Forward command will construct a new Snake command of the ordinal indicated in the Forward Header. It will connect to the target node in a new session, construct the Command Header, and send the encrypted command parameter data on to the target node. The parameter data already will have been encrypted using the key associated with the target node, so that the target implant will be able to decrypt the parameter data and execute the command.

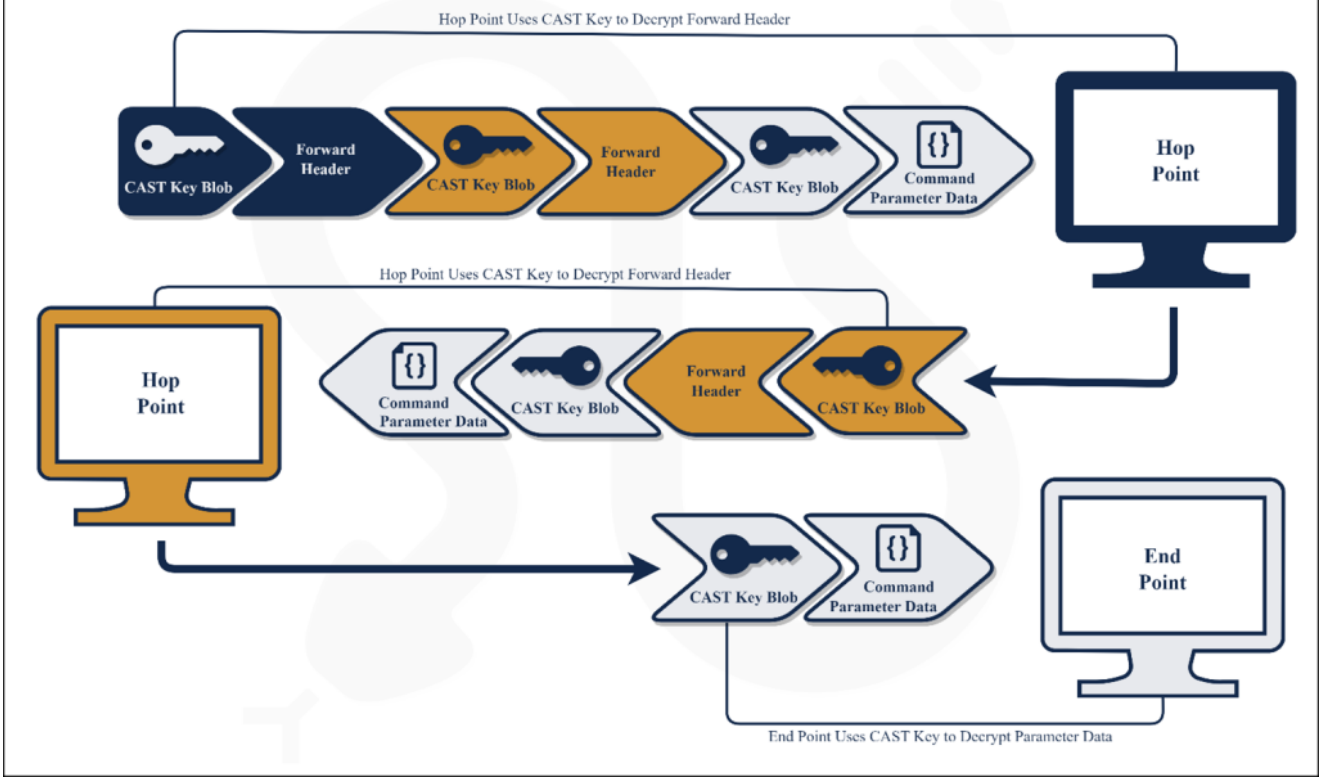
When the Forward command is constructed, the CAST key used to CAST-128 encrypt the payload data—to include the 0x199-byte header and the parameter data to be forwarded—is encrypted with the RSA key pair used by the first implant. The parameter data that contains the parameters for the command to be forwarded is also CAST-128 encrypted, but the key used to encrypt the parameter data is encrypted with the RSA key pair used by the target node. The first implant knows through the header what command ordinal it is forwarding, but it is unable to decrypt the parameter data.

If the Forward Header sent to the first implant indicates that the command to be forwarded was another Forward command, the first target node will decrypt the parameter data and find another Forward Header. This first target node implant will then go through the same process to connect to the next target node, constructing the new command with the ordinal indicated in the second Forward Header to send the remaining encrypted parameter data to the next target node. This will repeat until the command to be forwarded is something other than another Forward command.

The Command Header and pertinent parameters for each target node are encrypted specifically for that node by the operator before the Forward command is sent into the Snake P2P network. To illustrate, the diagram below shows how the buffer might look when several Forward commands are chained together to include two hop points and an end point. The first hop point (HP1) will recover the first CAST key and CAST-128 decrypt the rest of the buffer, which will uncover the first Forward Header. HP1 will then forward the remainder of the decrypted buffer to the next hop point (HP2), starting with the second CAST key blob. HP2 will recover the second CAST key and CAST-128 decrypt the rest of the buffer, which will uncover the second Forward Header. HP2 will then forward the remainder of the decrypted buffer to the end point, starting with the third CAST key blob. The end point will recover the CAST key, decrypt the command parameter data, and execute the command.

When a target machine has executed a forwarded command, the return data is encrypted with that implant's RSA keys and returned directly to the previous hop point. As the data is returned up the chain in the Snake P2P network, the intermediate hop points do not manipulate the encrypted data, as they do not have the RSA private key necessary to do so. In this manner, the return data is de-facto end-to-end encrypted throughout the P2P network until it arrives back at the FSB operator.

Figure 10: Forward Command Structure



SNAKE IMPLANT OPERATION

Snake uses two main methods for communication and command execution, namely Passive and Active. In general, Snake operators will employ Active operations to communicate with hop points within Snake's infrastructure; however, hop points can and do sometimes operate using Snake's Passive method. Snake's end points tend to solely operate using the Passive method.

Active Operations

During Active operations, Snake commands are issued by an FSB operator or a script to a target machine, generally through Forward commands (described in the previous section). The response to the command is immediately returned to the point of origin following the same path that the command took to reach its end target, as shown in the previous figure on Forward command structure.

Passive Operations

During Passive operations, Snake implants operate on their own, without the synchronous interaction of FSB operators. The nodes with which an implant communicates during Passive operations are stored within its 0x2 Container(s) as communication channels. Up to ten communication channels can be present at any time; an operator can change these channels via the Set Config Item command.

Passive Intake

During Passive operations, the implant will beacon by sending a Queue Read (0xb) command to one of its stored communication channels that it has chosen at random. These Queue Read commands look for a Queue Item within a Container with an Instance Number equal to the implant's UID. The matching UID indicates the Queue Items in this Container are intended for the beaconing implant.

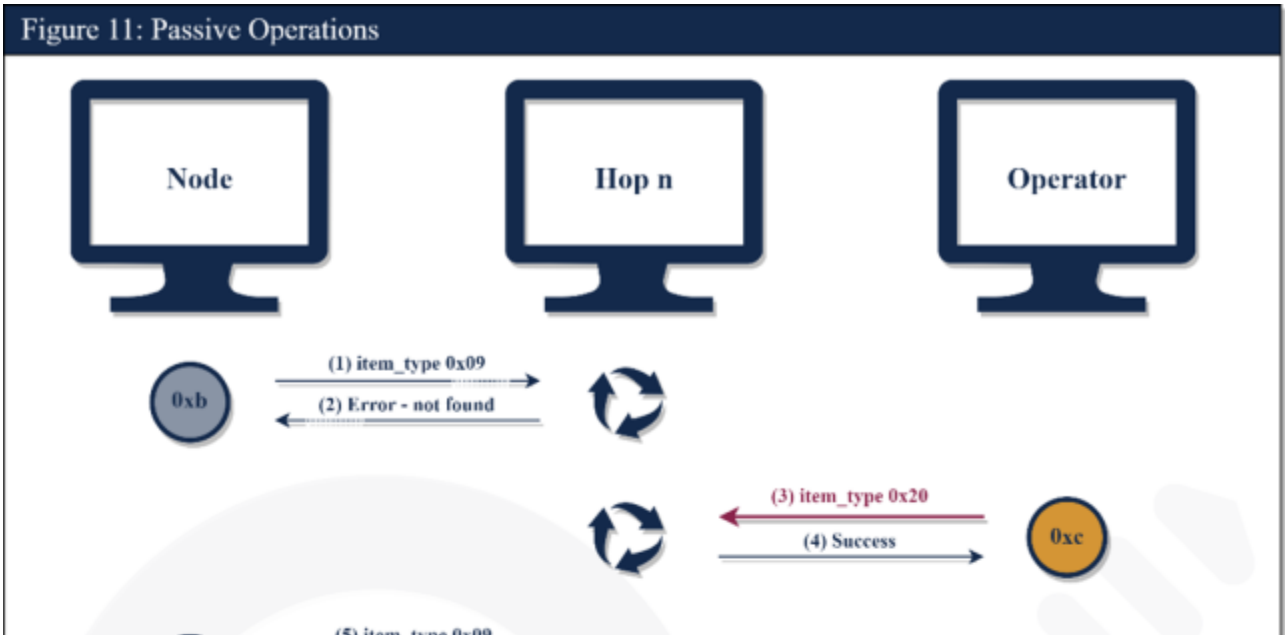
If such a Queue Item is found, the beaconing implant will read in the Queue Item and delete it off of the host from which it was read. There can be multiple Queue Items found within the specified Queue Container that was beaconed to; each Queue Read command will read one of these items. This process is repeated until all items within the Container are read, which the infrastructure node will indicate by sending a specific error in response to the Queue Read. This beaconing will continue to randomly select hosts at nondeterministic time intervals for as long as the implant is set to perform Passive operations.

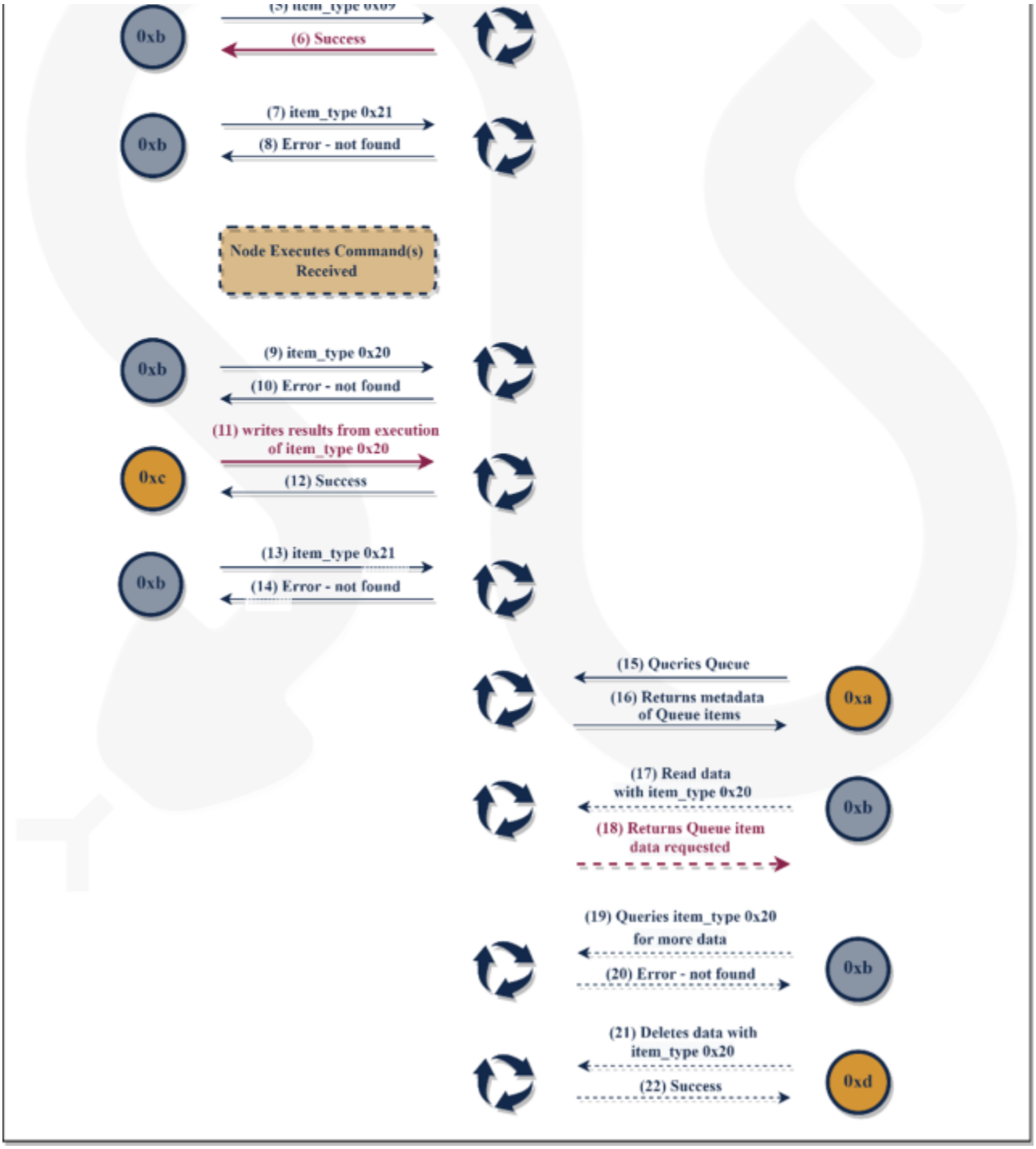
Passive Data Exfiltration

Similar to how Snake intakes commands passively, it can also exfiltrate the resulting data passively. This is done using Queue Write (0xc) commands to write to one of the stored communication channels chosen at random. Once the data is off the end point node, operators generally retrieve it manually or using a script. The Item Type field, which is unique per executed Snake command, is needed to associate the exfiltrated data with the target node on which the command was run.

In the context of Passive Snake communications, the term Item Type is defined as a UID for a given Snake command and its resulting data. The Item Type serves as a unique identifier to associate the results of command execution with the original command written by the operator. When the FSB collects the data, Snake knows exactly what infection the data came from, and therefore it can determine what key to use to successfully decrypt the data.

Figure 11: Passive Operations





To illustrate how Passive operations are conducted between the end points, the operator, and the hop points in between, see the diagram above, which is explained further by the following steps:

- (1), (2): During Passive operations, the Node randomly chooses a host from amongst its stored communication channels and will beacon out to it with a Queue Read command (Hop Point 1 in this case). The Item Type for these beacons will be one greater than the Item Type of the last command received by the Node, indicating in this example that a command of Item Type 0x08 was the last command that was read in by the Node during Passive operations. This Node will continue to beacon with Item Type 0x09 until it receives a command, via Passive operations, with an Item Type of 0x09 or greater. The lines are dotted for (1) and (2) as this activity will be repeated at random intervals until a successful Queue Read occurs.
- (3), (4): In these steps, the operator uses a Queue Write command to write a command to Hop Point that is ultimately intended for the Node. The Item Type of the command being written to Hop Point 1 is assigned 0x20 (for this example). Note that the path of this command, its execution, and its results making it back to the operator can be tracked via the red text.
- (5), (6): The Node continues to beacon out looking for commands to read in (5). The return (6) is successful, and the command written by the operator to Hop Point 1 (3) is read in by the Node, then deleted from Hop Point 1.
- (7), (8): The Node attempts another Queue Read to Hop Point 1, however now the Item Type is set to 0x21, one greater than the command that was just read in by the Node at (5) and (6). This returns an error as Hop Point 1 has nothing else for the Node to read in, indicating to the Node that everything at Hop Point 1 was read.
- (9), (10): At this point, the Node has executed the command it read in at (5) and (6) and is attempting to send back the results. The Node randomly selects another host from its stored communication channels, Hop Point 2 in this case, and sends out a 0xb command to make sure that the Item Type 0x20, the Item Type of the command it executed, does not already exist within the Queue of Hop Point 2. If it receives an error, there is no Item with Item Type 0x20 on Hop Point 2, and the Node can proceed to send the command results.
- (11), (12): Here the data from the executed command is written to Hop Point 2 with Item Type 0x20 into its 0x1 Container with a 0xc command, the Item Type the command was initially given at creation (3).
- (13), (14): The Node continues its normal beaconing routine again as seen in (1) and (2), searching for Item Type 0x21, one greater than the Item Type of the most recently executed command. As in (1) and (2), the lines here are dotted to denote that this process will repeat until there was a successful beacon as in (5) and (6).

- (15-22): These steps show how the operator retrieves the resulting data that was written to Hop Point 2. The Queue Enumerate command (15) lists the contents of Hop Point 2's 0x1 Container, showing the data written by the Node (11). This data is identifiable by its Item Type, namely 0x20. The Queue Read command (17) reads in the Item that was found in Hop Point 2's Container. The Queue Read command that follows (19) is asking if there is any data left. In this case, the entirety of the data was read with the first Queue Read (17, 18). Therefore, the error returned from second Queue Read command (20) lets the operator know all of the data from Item Type 0x20 was read and there is nothing further. A Queue Delete command (21) follows and is sent to delete the item with Item Type 0x20 from Hop Point 2.
- The subsequent Queue Read, Queue Read, and Queue Delete commands (17-21) are denoted with dashed lines to indicate that this sequence of commands is repeated for all items returned from the Queue Enumerate command (15).

MITIGATIONS

A number of complementary detection techniques effectively identify some of the more recent variants of Snake. However, as described above, Snake is purpose-built to avoid large-scale detection. Below is a discussion of the advantages and disadvantages of various detection methodologies available for Snake.

Note that some of the techniques identified in this section can affect the availability or stability of a system. Defenders should follow organizational policies and incident response best practices to minimize the risk to operations while hunting for Snake.

Network-Based Detection

Network Intrusion Detection Systems (NIDS) can feasibly identify some of the more recent variants of Snake and its custom network protocols as detailed above.

Advantages: High-confidence, large-scale (network-wide) detection of custom Snake communication protocols.

Disadvantages: Low visibility of Snake implant operations and encrypted data in transit. **There is some potential for false positives in the Snake http, http2, and tcp signatures.** Snake operators can easily change network-based signatures.

Snake http

Snake client-to-server http and http2 traffic is contained within an arbitrary HTTP header field. The header field value for http begins with 10 pure alphanumeric characters, followed by base64 encoding of 8 bytes, which yields exactly 11 valid base64 characters plus one base64 padding character.

```
^[0-9A-Za-z]{10}[0-9A-Za-z/\+]{11}=
```

The following two Suricata rules will detect the traffic described:

```
alert http any any -> any any (msg: "http rule (Cookie)";\
    pcre:"/[0-9A-Za-z]{10}[0-9A-Za-z\/\+]{11}=/C";\
    flow: established, to_server;\
    sid: 7; rev: 1;)
alert http any any -> any any (msg: "http rule (Other Header)";\
    pcre:"/[0-9A-Za-z]{10}[0-9A-Za-z\/\+]{11}=/H";\
    flow: established, to_server;\
    sid: 8; rev: 1;)
```

Snake http2

The header field value for http2 begins with 22 pure alphanumeric characters (base62 with non-extraneous characters), followed by the base62 encoding of at least 8 bytes, which must comprise at least 11 base62 characters with the four extraneous characters allowed. The actual requirement is stricter than this expression, since the total number of non-extraneous characters alone must equal or exceed 11; however, it is not possible to encode that aspect into a regular language.

```
^[0-9A-Za-z]{22}[0-9A-Za-z/;_=]{11}
```

The following two Suricata rules will detect the traffic described:

```
alert http any any -> any any (msg: "http2 rule (Cookie)";\
    pcre:"/[0-9A-Za-z]{22}[0-9A-Za-z\/_=\;]{11}/C";\
    flow: established, to_server;\
    sid: 9; rev: 1;)
alert http any any -> any any (msg: "http2 rule (Other Header)";\
    pcre:"/[0-9A-Za-z]{22}[0-9A-Za-z\/_=\;]{11}/H";\
    flow: established, to_server;\
    sid: 10; rev: 1;)
```

Snake tcp

The client-to-server communication for tcp must begin with the ustart, which is not captured in this signature set. Immediately following the ustart, the next client-to-server communication must be the big-endian 32-bit unsigned integer 8 followed by any 8 bytes of data. The next communication must also be client-to-server, and it must comprise the big-endian 32-bit unsigned integer 4 followed by any 4 bytes of data. The next two communications must be server-to-client, comprising the integer 8 followed by 8 bytes of data and the integer 4 followed by 4 bytes of data.

The following six Suricata rules will, in conjunction, detect traffic of the form described:

```
alert tcp any any -> any any (msg: "tcp rule";\
    content: "|00 00 00 08|"; startswith; dsize: 12;\
    flow: established, to_server; flowbits: set, a8; flowbits: noalert;\
    sid: 1; rev: 1;)
alert tcp any any -> any any (msg: "tcp rule";\
    content: "|00 00 00 04|"; startswith; dsize:8;\
    flow: established, to_server; flowbits: isset, a8; flowbits: unset, a8;\
    flowbits: set, a4; flowbits: noalert;\
    sid: 2; rev: 1;)
alert tcp any any -> any any (msg: "tcp rule";\
    content: "|00 00 00 08|"; startswith; dsize: 4;\
    flow: established, to_client; flowbits: isset, a4; flowbits: unset, a4;\
    flowbits: set, b81; flowbits: noalert;\
    sid: 3; rev: 1;)
alert tcp any any -> any any (msg: "tcp rule";\
    dsize: 8; flow: established, to_client; flowbits: isset, b81;\
    flowbits: unset, b81; flowbits: set, b8; flowbits: noalert;\
    sid: 4; rev: 1;)
alert tcp any any -> any any (msg: "tcp rule";\
    content: "|00 00 00 04|"; startswith; dsize: 4;\
    flow: established, to_client; flowbits: isset, b8; flowbits: unset, b8;\
    flowbits: set, b41; flowbits: noalert;\
    sid: 5; rev: 1;)
alert tcp any any -> any any (msg: "tcp rule";\
    dsize: 4; flow: established, to_client; flowbits: isset, b41;\
    flowbits: unset, b41;\
    sid: 6; rev: 1;)
```

Host-Based Detection

Advantages: High confidence based on totality of positive hits for host-based artifacts.

Disadvantages: Many of the artifacts on the host are easily shifted to exist in a different location or with a different name. As the files are fully encrypted, accurately identifying these files is difficult.

Covert Store Detection

The Snake covert store comprises a file-backed NTFS (usually) or FAT-16 (rarely) filesystem. The filesystem is encrypted with CAST-128 in CBC mode. The encryption key can be either statically hardcoded or dynamically stored in a specified Windows registry location. The IV is 8 bytes, since CAST-128 has an 8-byte block length. The first byte of the IV for any 512-byte block of the covert store is the 0-indexed block number. The remaining bytes of the IV are the corresponding bytes of the key, meaning that bytes at 0-indexed indices 1 through 7 of the IV are the bytes at 0-indexed indices 1 through 7 of the key.

```
When statically hardcoded, the encryption key has the following constant value:

     A1 D2 10 B7 60 5E DA 0F A1 65 AF EF 79 C3 66 FA
```

When stored in the Windows registry, the encryption key is the classname associated with the following key:

```
SECURITY\Policy\Secrets\n
```

The following initial 8-byte sequences are known to be used by NTFS or FAT-16 filesystems as observed:

```
EB 52 90 4E 54 46 53 20
EB 5B 90 4E 54 46 53 20
EB 3C 90 4D 53 44 4F 53
EB 00 00 00 00 00 00 00
```

For tool development, the following test vector illustrates the encryption of the first given header above (EB 52 90 …) using CAST-128 with the default key shown above and the IV constructed as described, given this header occurs at the beginning of the first 512-byte block of the covert store.

Plaintext: EB 52 90 4E 54 46 53 20
Key: A1 D2 10 B7 60 5E DA 0F A1 65 AF EF 79 C3 66 FA
IV: 00 D2 10 B7 60 5E DA 0F
Ciphertext: C2 C7 F4 CA F7 DA 3A C8

By encrypting each possible initial filesystem byte sequence with CAST-128 using the key obtained from the registry—or the default encryption key if the registry entry does not exist—and searching for any file with a size that is an even multiple of 220, it is possible to efficiently detect Snake covert stores. Validation can be performed by decrypting the entire file using the outlined methodology and then verifying that it comprises an NTFS or FAT-16 filesystem.

Other On-Disk Artifact Detection

Registry Blob

The registry blob is generally found at the location listed below. In case it is not present at its typical location, the registry blob can be found by searching the full registry for a value of at least 0x1000 bytes in size and entropy of at least 7.9.

Typical Name: Unknown (RegBlob)
Typical Path: HKLM\SOFTWARE\Classes\.wav\OpenWithProgIds
Characteristics: High Entropy

Queue File

Typical Name: < RANDOM_GUID >.<RANDOM_GUID>.crmlog
Typical Path: %windows\registration\
Unique Characteristics: High Entropy, file attributes of hidden, system, and archive

Role: Snake Queue File

The Snake Queue File generally has a predictable path and filename structure, in addition to being high entropy. The Snake Queue File can be located by scanning all files in the typical queue path with filenames matching a regular expression that captures the typical naming convention. Files meeting these criteria should be scanned for high entropy, which is performed by the Yara rule below:

```
rule HighEntropy
{
    meta:
        description = "entropy rule"

    condition:
        math.entropy(0, filesize) >= 7.0
}
```

The following UNIX find command will scan files with names matching the GUID-based convention (note that the HighEntropy yara rule is assumed to be contained in a file named “1.yar”):

```
find /PATH/TO/WINDOWS_DIR -type f -regextype posix-egrep -iregex \
    '.*\/registration/(\{[0-9A-F]{8}\-([0-9A-F]{4}\-){3}[0-9A-F]{12}\}\.){2}crmlog' \
     -exec yara 1.yar {} \;
```

The following PowerShell command does the same:

```
Get-ChildItem -Recurse -File -Path %WINDOWS% | Where-Object {
  $_.FullName -match
  '(?i)/registration/(\{[0-9A-F]{8}\-([0-9A-F]{4}\-){3}[0-9A-F]{12}\}\.){2}crmlog$'
} | ForEach-Object {
  yara 1.yar $_.FullName
}
```

Comadmin

Typical Name: comadmin.dat
Typical Path: %windows%\system32\Com
Unique Characteristics: High Entropy
Role: Houses Snake’s kernel driver and the driver’s loader

The Snake Comadmin file can be found using analogous techniques to that presented above for locating the Snake Queue File. The following UNIX find command will do so:

```
find /PATH/TO/WINDOWS -type f -regextype posix-egrep -iregex \
    '.*\/system32/Com/comadmin\.dat' \
     -exec yara 1.yar {} \;
```

The following PowerShell command does the same:

```
Get-ChildItem -Recurse -File -Path %WINDOWS% | Where-Object {
    $_.FullName -match '(?i)/system32/Com/comadmin\.dat$'
} | ForEach-Object {
    yara 1.yar $_.FullName
}
```

Werfault

Typical Name: Werfault.exe
Typical Path: %windows%\WinSxS\x86_microsoft-windows-errorreportingfaults_31bf3856ad364e35_4.0.9600.16384_none_a13f7e283339a0502\
Unique Characteristics: Icon is different than that of a valid Windows Werfault.exe file
Role: Persistence mechanism

The Snake Werfault.exe file has non-standard icon sizes, which form the basis of the Yara rule below. This rule should be run on all files in the typical path, specifically the %Windows%\WinSxS directory.

```
rule PeIconSizes
{
    meta:
        description = "werfault rule"

    condition:
        pe.is_pe
        and
        for any rsrc in pe.resources:
            (rsrc.type == pe.RESOURCE_TYPE_ICON and rsrc.length == 3240)
        and
        for any rsrc in pe.resources:
            (rsrc.type == pe.RESOURCE_TYPE_ICON and rsrc.length == 1384)
        and
        for any rsrc in pe.resources:
            (rsrc.type == pe.RESOURCE_TYPE_ICON and rsrc.length == 7336)
}
```

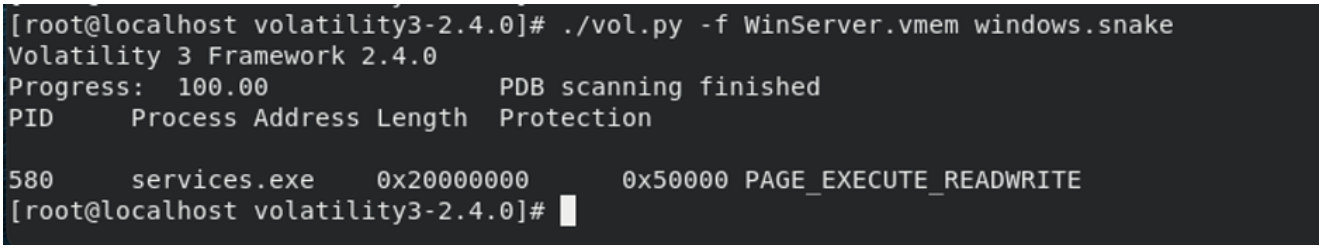
Memory Analysis

Advantages: High confidence as memory provides the greatest level of visibility into Snake's behaviors and artifacts.

Disadvantages: Potential impact on system stability, difficult scalability.

Capturing and analyzing the memory of a system will be the most effective approach in detecting Snake because it bypasses many of the behaviors that Snake employs to hide itself. With a memory analysis tool, such as Volatility, detection of a Snake compromise may be possible.
Snake's principal user mode component is injected into a chosen process via a single

allocation of PAGE_EXECUTE_READWRITE memory. The starting offset is generally 0x20000000, however the module does allow for relocation if needed. Additionally, since the user mode component is not obfuscated in any way, a valid PE header can be located at the beginning of the allocated memory region. Further validation can be performed by confirming the presence of strings known to exist in the user mode component also within the memory region. A plugin compatible with Volatility3 which can scan all processes on a system using this method is provided in the Appendix. A screenshot showing the results of the plugin successfully detecting Snake is displayed below.



PREVENTION

Note that the mitigations that follow are not meant to protect against the initial access vector and are only designed to prevent Snake's persistence and hiding techniques.

Change Credentials and Apply Updates

System owners who are believed to be compromised by Snake are advised to change their credentials immediately (from a non-compromised system) and to not use any type of passwords similar to those used before. Snake employs a keylogger functionality that routinely returns logs back to FSB operators. Changing passwords and usernames to values which cannot be brute forced or guessed based on old passwords is recommended.

System owners are advised to apply updates to their Operating Systems. Modern versions of Windows, Linux, and MacOS make it much harder for adversaries to operate in the kernel space. This will make it much harder for FSB actors to load Snake's kernel driver on the target system.

Execute Organizational Incident Response Plan

If system owners receive detection signatures of Snake implant activity or have other indicators of compromise that are associated with FSB actors using Snake, the impacted organization should immediately initiate their documented incident response plan.

We recommend implementing the following Cross-Sector Cybersecurity Performance Goals (CPGs) to help defend against FSB actors using Snake, or mitigate negative impacts post-compromise:

CPG 2.A: Changing Default Passwords will prevent FSB actors from compromising default credentials to gain initial access or move laterally within a network.

CPG 2.B: Requiring Minimum Password Strength across an organization will prevent FSB actors from being able to successfully conduct password spraying or cracking operations.

CPG 2.C: Requiring Unique Credentials will prevent FSB actors from compromising valid accounts through password spraying or brute force.

CPG 2.E Separating User and Privileged Accounts will make it harder for FSB actors to gain access to administrator credentials.

CPG 2.F. Network Segmentation to deny all connections by default unless explicitly required for specific system functionality, and ensure all incoming communication is going through a properly configured firewall.

CPG 2.H Implementing Phishing Resistant MFA adds an additional layer of security even when account credentials are compromised and can mitigate a variety of attacks towards valid accounts, to include brute forcing passwords and exploiting external remote services software.

CPG 4.C. Deploy Security.txt Files to ensure all public facing web domains have a security.txt file that conforms to the recommendations in RFC 9118.

APPENDIX

Partnership

This advisory was developed as a joint effort by an international partnership of multiple agencies in furtherance of the respective cybersecurity missions of each of the partner agencies, including our responsibilities to develop and issue cybersecurity specifications and mitigations. This partnership includes the following organizations:

Collectively, we use a variety of sources, methods, and partnerships to acquire information about foreign cyber threats. This advisory contains the information we have concluded can be publicly released, consistent with the protection of sources and methods and the public interest.

Disclaimer

The information in this report is being provided “as is” for informational purposes only. We do not endorse any commercial product or service, including any subjects of analysis. Any reference to specific commercial products, processes, or services by service mark,

trademark, manufacturer, or otherwise, does not constitute or imply endorsement, recommendation, or favoring by co-authors.

MITRE ATT&CK Techniques

This advisory uses the MITRE ATT&CK® for Enterprise framework, version 13. See MITRE ATT&CK for Enterprise for all referenced tactics and techniques. MITRE and ATT&CK are registered trademarks of The MITRE Corporation. This report references the following MITRE ATT&CK techniques.

Technique Title	ID	Use
Network Connection Enumeration	T0840	Adversaries may perform network connection enumeration to discover information about device communication patterns.
Data Obfuscation	T1001	Adversaries may obfuscate command and control traffic to make it more difficult to detect.
Protocol Impersonation	T1001.003	Adversaries may impersonate legitimate protocols or web service traffic to disguise command and control activity and thwart analysis efforts.
OS Credential Dumping	T1003	Adversaries may attempt to dump credentials to obtain account login and credential material, normally in the form of a hash or a clear text password, from the operating system and software.
Rootkit	T1014	Adversaries may use rootkits to hide the presence of programs, files, network connections, services, drivers, and other system components.
Obfuscated Files or Information	T1027	Adversaries may attempt to make an executable or file difficult to discover or analyze by encrypting, encoding, or otherwise obfuscating its contents on the system or in transit.
Software Packing	T1027.002	Adversaries may perform software packing or virtual machine software protection to conceal their code.

Masquerading	T1036	Adversaries may attempt to manipulate features of their artifacts to make them appear legitimate or benign to users and/or security tools.
Network Sniffing	T1040	Adversaries may sniff network traffic to capture information about an environment, including authentication material passed over the network.
Network Service Discovery	T1046	Adversaries may attempt to get a listing of services running on remote hosts and local network infrastructure devices, including those that may be vulnerable to remote software exploitation.
Dynamic-link Library Injection	T1055.001	Adversaries may inject dynamic-link libraries (DLLs) into processes in order to evade process-based defenses as well as possibly elevate privileges.
Keylogging	T1056.001	Adversaries may log user keystrokes to intercept credentials as the user types them.
PowerShell	T1059.001	Adversaries may abuse PowerShell commands and scripts for execution.
Application Layer Protocol	T1071	Adversaries may communicate using OSI application layer protocols to avoid detection/network filtering by blending in with existing traffic.
Web Protocols	T1071.001	Adversaries may communicate using application layer protocols associated with web traffic to avoid detection/network filtering by blending in with existing traffic.
Mail Protocols	T1071.003	Adversaries may communicate using application layer protocols associated with electronic mail delivery to avoid detection/network filtering by blending in with existing traffic.
DNS	T1071.004	Adversaries may communicate using the Domain Name System (DNS) application layer protocol to avoid detection/network filtering by blending in with existing traffic.

Data Staged	T1074	Adversaries may stage collected data in a central location or directory prior to Exfiltration.
Valid Accounts	T1078	Adversaries may obtain and abuse credentials of existing accounts as a means of gaining Initial Access, Persistence, Privilege Escalation, or Defense Evasion.
File and Directory Discovery	T1083	Adversaries may enumerate files and directories or may search in specific locations of a host or network share for certain information within a file system.
Multi-hop Proxy	T1090.003	To disguise the source of malicious traffic, adversaries may chain together multiple proxies.
Non-Application Layer Protocol	T1095	Adversaries may use an OSI non-application layer protocol for communication between host and C2 server or among infected hosts within a network.
Multi-Stage Channels	T1104	Adversaries may create multiple stages for command and control that are employed under different conditions or for certain functions.
Native API	T1106	Adversaries may interact with the native OS application programming interface (API) to execute behaviors.
Modify Registry	T1112	Adversaries may interact with the Windows Registry to hide configuration information within Registry keys, remove information as part of cleaning up, or as part of other techniques to aid in persistence and execution.
Automated Collection	T1119	Once established within a system or network, an adversary may use automated techniques for collecting internal data.
Data Encoding	T1132	Adversaries may encode data to make the content of command and control traffic more difficult to detect.

Non-Standard Encoding	T1132.002	Adversaries may encode data with a non-standard data encoding system to make the content of command and control traffic more difficult to detect.
Network Share Discovery	T1135	Adversaries may look for folders and drives shared on remote systems as a means of identifying sources of information to gather as a precursor for Collection and to identify potential systems of interest for Lateral Movement.
Deobfuscate/Decode Files or Information	T1140	Adversaries may use Obfuscated Files or Information to hide artifacts of an intrusion from analysis.
Exploit Public-Facing Application	T1190	Adversaries may attempt to exploit a weakness in an Internet-facing host or system to initially access a network.
Domain Trust Discovery	T1482	Adversaries may attempt to gather information on domain trust relationships that may be used to identify lateral movement opportunities in Windows multi-domain/forest environments.
Installer Packages	T1546.016	Adversaries may establish persistence and elevate privileges by using an installer to trigger the execution of malicious content.
Dynamic Linker Hijacking	T1547.006	Adversaries may execute their own malicious payloads by hijacking environment variables the dynamic linker uses to load shared libraries.
Inter-Process Communication	T1559	Adversaries may abuse inter-process communication (IPC) mechanisms for local code or command execution.
Archive Collected Data	T1560.003	An adversary may compress and/or encrypt data that is collected prior to exfiltration.
Hide Artifacts	T1564	Adversaries may attempt to hide artifacts associated with their behaviors to evade detection.

Service Execution	T1569.002	Adversaries may abuse the Windows service control manager to execute malicious commands or payloads.
Lateral Tool Transfer	T1570	Adversaries may transfer tools or other files between systems in a compromised environment.
Protocol Tunneling	T1572	Adversaries may tunnel network communications to and from a victim system within a separate protocol to avoid detection/network filtering and/or enable access to otherwise unreachable systems.
Encrypted Channel	T1573	Adversaries may employ a known encryption algorithm to conceal command and control traffic rather than relying on any inherent protections provided by a communication protocol.
Symmetric Cryptography	T1573.001	Adversaries may employ a known symmetric encryption algorithm to conceal command and control traffic rather than relying on any inherent protections provided by a communication protocol.
Asymmetric Cryptography	T1573.002	Adversaries may employ a known asymmetric encryption algorithm to conceal command and control traffic rather than relying on any inherent protections provided by a communication protocol.
DLL Side-Loading	T1574.002	Adversaries may execute their own malicious payloads by side-loading DLLs.
Compromise Infrastructure	T1584	Adversaries may compromise third-party infrastructure that can be used during targeting.
Malware	T1587.001	Adversaries may develop malware and malware components that can be used during targeting.
Obtain Capabilities	T1588	Adversaries may buy and/or steal capabilities that can be used during targeting.
Stage Capabilities	T1608	Adversaries may upload, install, or otherwise set up capabilities that can be used during targeting.

Deploy Container	T1610	Adversaries may deploy a container into an environment to facilitate execution or evade defenses.

Volatility Plugin

The following plugin for the Volatility memory analysis framework will scan all processes on the system until it finds the Snake user mode component injected into a process. If found, the plugin will list both the injected process and the virtual memory address at which the Snake user mode component is loaded.

```
# This plugin to identify the injected usermode component of Snake is based
# on the malfind plugin released with Volatility3
#
# This file is Copyright 2019 Volatility Foundation and licensed under the
# Volatility Software License 1.0
# which is available at https://www.volatilityfoundation.org/license/vsl-v1.0
import logging
from typing import Iterable, Tuple
from volatility3.framework import interfaces, symbols, exceptions, renderers
from volatility3.framework.configuration import requirements
from volatility3.framework.objects import utility
from volatility3.framework.renderers import format_hints
from volatility3.plugins.windows import pslist, vadinfo
vollog = logging.getLogger(__name__)
class snake(interfaces.plugins.PluginInterface):
    _required_framework_version = (2, 4, 0)

    @classmethod
    def get_requirements(cls):
        return [
            requirements.ModuleRequirement(name = 'kernel',
            description = 'Windows kernel',
            architectures = ["Intel32", "Intel64"]),
            requirements.VersionRequirement(name = 'pslist',
            component = pslist.PsList, version = (2, 0, 0)),
            requirements.VersionRequirement(name = 'vadinfo',
            component = vadinfo.VadInfo, version = (2, 0, 0))]

    @classmethod
    def list_injections(
            cls, context: interfaces.context.ContextInterface,
            kernel_layer_name: str, symbol_table: str,
            proc: interfaces.objects.ObjectInterface) -> Iterable[
            Tuple[interfaces.objects.ObjectInterface, bytes]]:
        proc_id = "Unknown"
        try:
            proc_id = proc.UniqueProcessId
            proc_layer_name = proc.add_process_layer()
        except exceptions.InvalidAddressException as excp:
            vollog.debug("Process {}: invalid address {} in layer {}".
            format(proc_id, excp.invalid_address, excp.layer_name))
            return
        proc_layer = context.layers[proc_layer_name]
        for vad in proc.get_vad_root().traverse():
            protection_string = vad.get_protection(vadinfo.VadInfo.
            protect_values(context, kernel_layer_name, symbol_table),
            vadinfo.winnt_protections)
            if not "PAGE_EXECUTE_READWRITE" in protection_string:
                continue

            if (vad.get_private_memory() == 1
                    and vad.get_tag() == "VadS") or (vad.get_private_memory()
```

```
                == 0 and protection_string !=
                "PAGE_EXECUTE_WRITECOPY"):
            data = proc_layer.read(vad.get_start(),
            vad.get_size(), pad = True)
            if data.find(b'\x4d\x5a') != 0:
                continue
            yield vad, data

    def _generator(self, procs):
        kernel = self.context.modules[self.config['kernel']]
        is_32bit_arch = not symbols.symbol_table_is_64bit(self.context,
        kernel.symbol_table_name)
        for proc in procs:
            process_name = utility.array_to_string(proc.ImageFileName)
            for vad, data in self.list_injections(self.context,
            kernel.layer_name, kernel.symbol_table_name, proc):
                strings_to_find = [b'\x25\x73\x23\x31',b'\x25\x73\x23\x32',
                b'\x25\x73\x23\x33',b'\x25\x73\x23\x34',
                b'\x2e\x74\x6d\x70', b'\x2e\x73\x61\x76',
                b'\x2e\x75\x70\x64']
                if not all(stringToFind in data for
                stringToFind in strings_to_find):
                    continue
                yield (0, (proc.UniqueProcessId, process_name,
                format_hints.Hex(vad.get_start()),
                           format_hints.Hex(vad.get_size()),
                           vad.get_protection(
                               vadinfo.VadInfo.protect_values(self.context,
                kernel.layer_name, kernel.symbol_table_name),
                vadinfo.winnt_protections)))
                return

    def run(self):
        kernel = self.context.modules[self.config['kernel']]
        return renderers.TreeGrid([("PID", int), ("Process", str),
        ("Address", format_hints.Hex), ("Length", format_hints.Hex),
        ("Protection", str)], self._generator(pslist.PsList.list_processes(
        context = self.context, layer_name = kernel.layer_name,
        symbol_table = kernel.symbol_table_name)))
```

Related Advisories

May 24, 2023

Cybersecurity Advisory | AA23-144a

People's Republic of China State-Sponsored Cyber Actor Living off the Land to Evade Detection

May 16, 2023

Cybersecurity Advisory | AA23-136A

#StopRansomware: BianLian Ransomware Group

May 11, 2023

Cybersecurity Advisory | AA23-131A

Malicious Actors Exploit CVE-2023-27350 in PaperCut MF and NG

Apr 18, 2023

Cybersecurity Advisory | AA23-108

APT28 Exploits Known Vulnerability to Carry Out Reconnaissance and Deploy Malware on Cisco Routers