DDoS

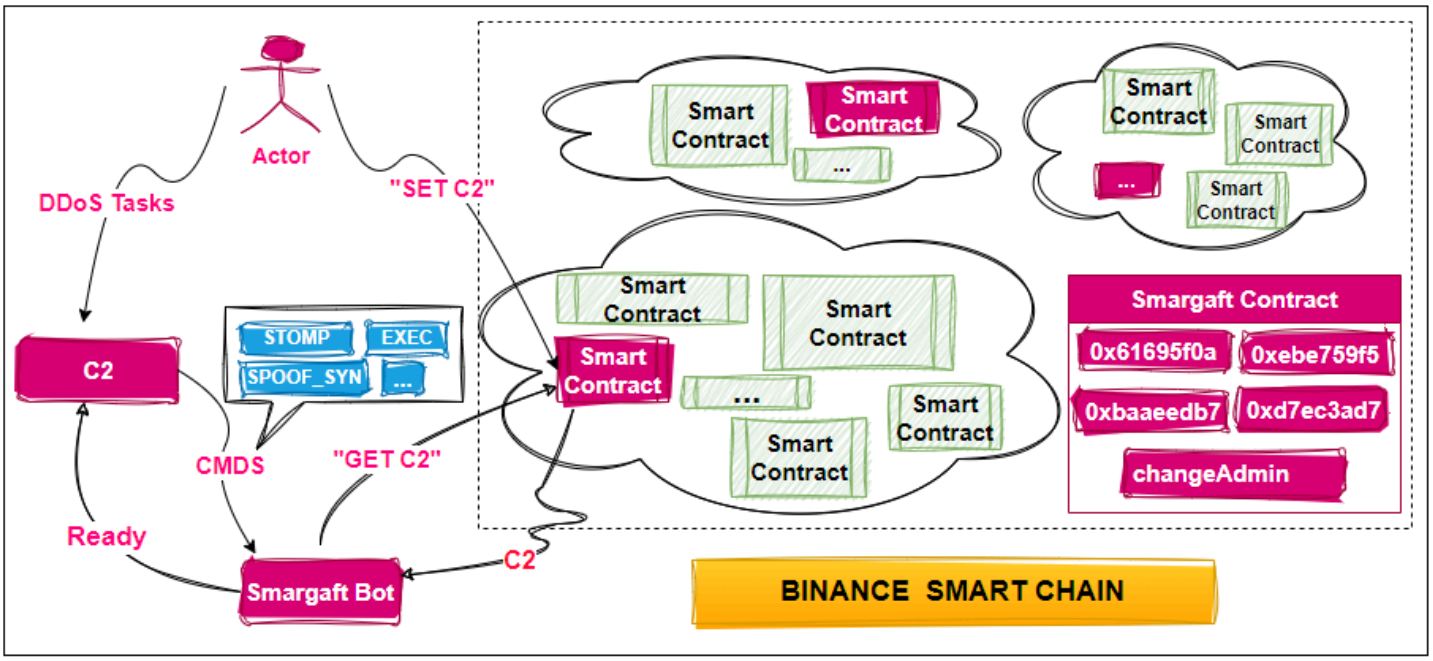
Smargaft Harnesses EtherHiding for Stealthy C2 Hosting

Alex.Turing, Acey9

2024年2月2日 • 16 min read

Background

At XLab, we see a lot of botnets every day, mainly tweaks of old Mirai and Gafgyt codes. These are pretty common and usually don't grab our attention. But today, we found something different. This new botnet uses some of Gafgyt's attack styles but has been built differently from the ground up. What's cool is that it uses the Binance Smart Chain to host commands and control server(C2), and infects shell scripts like a Virus to achieve persistence. When looking closer, we noticed that some antivirus vendors flagged this botnet as Mirai, which isn't right. Because of its smart use of contracts and Gafgyt's methods, we've decided to call it Smargaft. It mainly does DDoS attacks, runs system commands, and lets users connect anonymously using socks5 proxy.



The biggest highlight of Smargaft is its use of smart contracts for hosting its C2 infrastructure, a technique first disclosed in October 2023 and known in the industry as **EtherHiding**. It fully leverages the public and immutable nature of blockchain technology, making the "on-chain" C2 irremovable, which is a highly advanced and rare form of Bullet-Proof Hosting technique. This is the first time we've seen such technology applied in the botnet field. Another advantage of using smart contracts for cloud-based C2 configuration is flexibility. Malware authors can even design monitoring code to interact with smart contracts, enabling automatic updates of C2 when specific conditions are met or adjusting attack strategies based on environmental changes. Given that once abused, contracts can quickly become powerful tools for cybercrime, significantly increasing the difficulty of monitoring and management, we decided to write this article to share our latest findings with the community, hoping to help everyone more effectively identify and prevent these new types of cyber threats.

What is BSC, What is Smart Contract?

Binance Smart Chain (BSC) is a blockchain platform developed and maintained by Binance, launched in 2020. It's designed to support decentralized applications

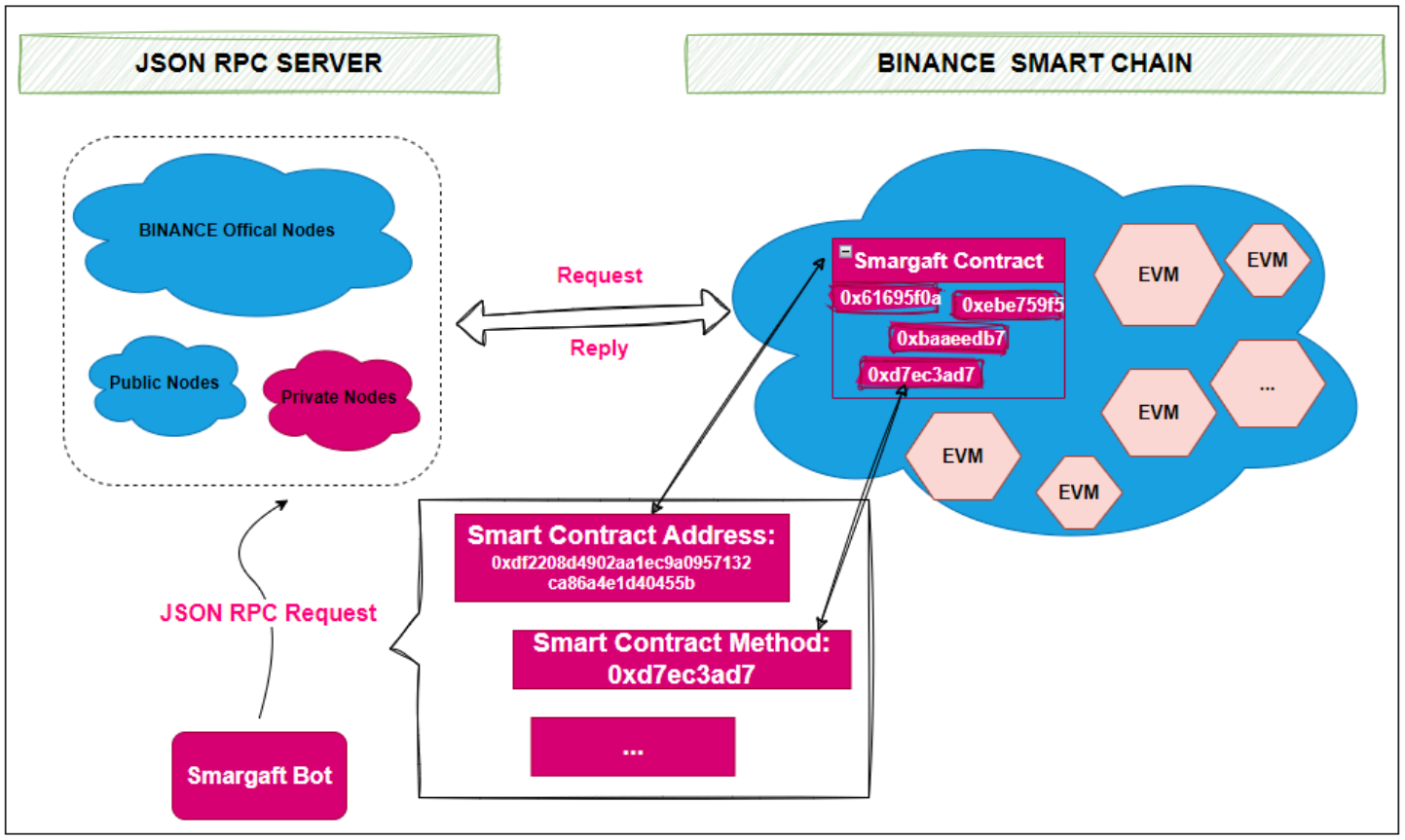
(DApps) and smart contracts, similar to Ethereum.

Smart contracts are automated protocols or programs executed on a blockchain. They consist of pre-written code designed to automatically perform actions or contractual terms under specific conditions. Smart contracts enable trustworthy transactions and interactions without intermediaries. When certain conditions are met, smart contracts can trigger various operations, such as transferring digital assets, distributing rewards, or creating tokens.

> *XLab notes:*
>
> > *Smart contracts are like special boxes placed on the blockchain. These boxes can store data, perform calculations, and allow people to view and utilize this data. One characteristic of the blockchain is that information is never deleted, so this box will always exist. Another feature of the blockchain is that all transactions are recorded, making operations conducted through this box traceable and immutable.*

In the scenario where "Smargaft utilizes smart contracts for C2 management," it essentially means that the developers of Smargaft configure and manage the C2 infrastructure through smart contracts, with the C2-related information ultimately stored on the blockchain. Once the C2 configuration is complete, Smargaft's malicious software (bot samples) communicates with the blockchain network via JSON RPC to retrieve C2 information.

Specifically, the bot samples send a request to an RPC server, which includes the address of the Smargaft smart contract and the contract function to be called. Upon receiving this request, the RPC server forwards it to the blockchain network. Then, the blockchain nodes receive the request and use the Ethereum Virtual Machine (EVM) to load and execute the specified function within the smart contract. After execution, the results are sent back along the same path to the bot sample.



Considering the methods of querying data on the blockchain and the inherent characteristics of the blockchain itself, the advantages of the "smart contract-hosted C2" technology are mainly reflected in three aspects of unblockable:

1. **Unblockable Access Channels**: Due to the diversity of RPC nodes, including official nodes, public nodes, and privately built nodes, it's difficult to cover all possible access paths with a blacklist. Taking Smargaft as an example, even if Binance's official nodes are blocked in China, there are still multiple public nodes available for use. Smargaft's bot samples are embedded with 14 different RPC server addresses, ensuring that at least some nodes are accessible.
2. **Unblockable Configuration Mechanism**: Malicious smart contracts deployed on the blockchain are difficult to directly ban or delete. For example, the smart contract address of Smargaft (e.g., "0xdf2208d4902aa1ec9a0957132ca86a4e1d40455b") is not subject to direct regulation or blocking by Binance or other blockchain operators.
3. **Unblockable C2 Information**: Once C2 information is stored on the blockchain via a smart contract, it cannot be deleted or altered. In the case of Smargaft, C2 information is permanently recorded on a specific

block of the Binance Smart Chain (e.g., block number 34731229), ensuring its persistence and immutability.

Smargaft Contracts

For analyzing smart contracts, the platform provided by Binance, BscScan, can be used for exploration and analysis. The contract address used in the Smargaft sample is **0xdf2208d4902aa1ec9a0957132ca86a4e1d40455b**. Based on the timeline we've compiled, it appears highly likely that the author of Smargaft is Russian, who began experimenting with the technology of smart contract-hosted C2 on September 15, 2023, completed testing by September 20, 2023, and officially deployed it on December 27, 2023.

```
+ 2023-09-15:
        + Transfer
        + Src Wallet: 0x7Be249AA69c631c7aa5De4F3aDbFb8A9db8DfD09
        + Dst Wallet: 0x16Cc46219d062257F384D85F84c7AbC7D9e34444
        + Amount: 0.06094994 BNB ($17.75)
+ 2023-09-15 :
        + Create the first contract (unstripped)
        + Wallet: 0x16Cc46219d062257F384D85F84c7AbC7D9e34444
        + Contract: 0xe77c6a0E10F2A469fb2afa667C99180E186233a8
        + Init:
                + Addr: 45.95.146.93
        + The comment is Russian
+ 2023-09-15:
        + Set address: 1.1.1.1
        + Wallet: 0x16Cc46219d062257F384D85F84c7AbC7D9e34444
        + Contract: 0xe77c6a0E10F2A469fb2afa667C99180E186233a8
+ 2023-09-15:
        + Create the second contract (same as the first one, but stripped)
        + Wallet: 0x16Cc46219d062257F384D85F84c7AbC7D9e34444
        + Contract: 0x862fbeb2456499a37e9146f1ef6eb5a57c2fb97a
        + InitAddr:
+ 2023-09-15:
        + Change the IP addresses around
                + 45.95.146.93 -> 10.202.30.40 -> 45.95.146.93 -> 1.22.33.44 -> 45.
        + Wallet: 0x16Cc46219d062257F384D85F84c7AbC7D9e34444
        + Contract: 0x862fbeb2456499a37e9146f1ef6eb5a57c2fb97a
+ 2023-09-20:
        + Create the third contract (add one variable and its set/get function, sti
        + Wallet: 0x16Cc46219d062257F384D85F84c7AbC7D9e34444
```

```
        + Contract: 0xdf2208d4902aa1ec9a0957132ca86a4e1d40455b
        + Init:
                + Addr: 45.95.146.93
                + New Parameter: hello
+ 2023-12-27
        + Set address: 45.95.146.93;94.103.188.167;185.132.125.193
        + Wallet: 0x16Cc46219d062257F384D85F84c7AbC7D9e34444
        + Contract: 0xdf2208d4902aa1ec9a0957132ca86a4e1d40455b
```

The original contract, **0xe77c6a0E10F2A469fb2afa667C99180E186233a8**, has its source code available for viewing, with many comments in Russian. It mainly functions to read and write the ServerAddr on the blockchain through **getServerAddr** and **setServerAddr**.

```
pragma solidity ^0.8.0;

contract ControlEbanataContract {
    string private serverAddr;
    address public admin;

    // Событие для уведомления о смене адреса сервера
    event ServerAddrChanged(string newAddr);

    constructor(string memory initialServerAddr) {
        serverAddr = initialServerAddr;
        admin = msg.sender;
    }

    // Изменить адрес сервера (только администратор)
    function setServerAddr(string memory newAddr) public {
        require(msg.sender == admin, "Only the admin can change the server address"
        serverAddr = newAddr;
        emit ServerAddrChanged(newAddr);
    }

    // Получить текущий адрес сервера
    function getServerAddr() public view returns (string memory) {
        return serverAddr;
    }

    // Изменить администратора (только текущий администратор)
    function changeAdmin(address newAdmin) public {
        require(msg.sender == admin, "Only the admin can change the admin");
        admin = newAdmin;
    }
}
```

After several tests, the final contract for Smargaft is **0xdf2208d4902aa1ec9a0957132ca86a4e1d40455b**. This version of the contract is in compiled bytecode.

Although the source code of the Smargaft contract is not directly viewable, it can be decompiled by clicking "Decompile Bytecode".

```
def storage:
  stor0 is array of struct at storage 0
  stor1 is array of struct at storage 1
  adminAddress is addr at storage 2

def unknown61695f0a(array _param1) payable:
{...}
 if _param1.length:
        stor1[].field_0 = Array(len=_param1.length, data=_param1[all])
{...}

def unknownd7ec3ad7() payable:
{...}
  return Array(len=stor1.length % 128, data=mem[128 len ceil32(stor1.length.field_1
{...}


def unknownebe759f5(array _param1) payable:
{...}
if _param1.length:
        stor0[].field_0 = Array(len=_param1.length, data=_param1[all])
{...}
def unknownbaaeedb7() payable:
{...}
  return Array(len=stor0.length % 128, data=mem[128 len ceil32(stor0.length.field_1
{...}
```

This is a straightforward contract application that uses **storage** to save variables `stor0`, `stor1`, and `adminAddress`. The function with identifier `0x61695f0a` saves the input `_param1` byte by byte into `stor1`, while `0xd7ec3ad7` reads the data from `stor1` and returns it as a string. The functions `0xebe759f5` and `0xbaaeedb7` operate similarly but target the variable `stor0`. Readers with programming experience will recognize these as a pair of **set/get** methods, allowing interaction with the contract to write or update data on the blockchain through this **set/get** mechanism.

For example, in actual operation, the author of Smargaft used the `0x61695f0a` method on `Dec-27-2023 09:55:26 PM +UTC` to write C2 data onto the blockchain.

In Smargaft's bot samples, the **eth_call** is used to invoke the smart contract's method **0xd7ec3ad7** to obtain the C2 information.

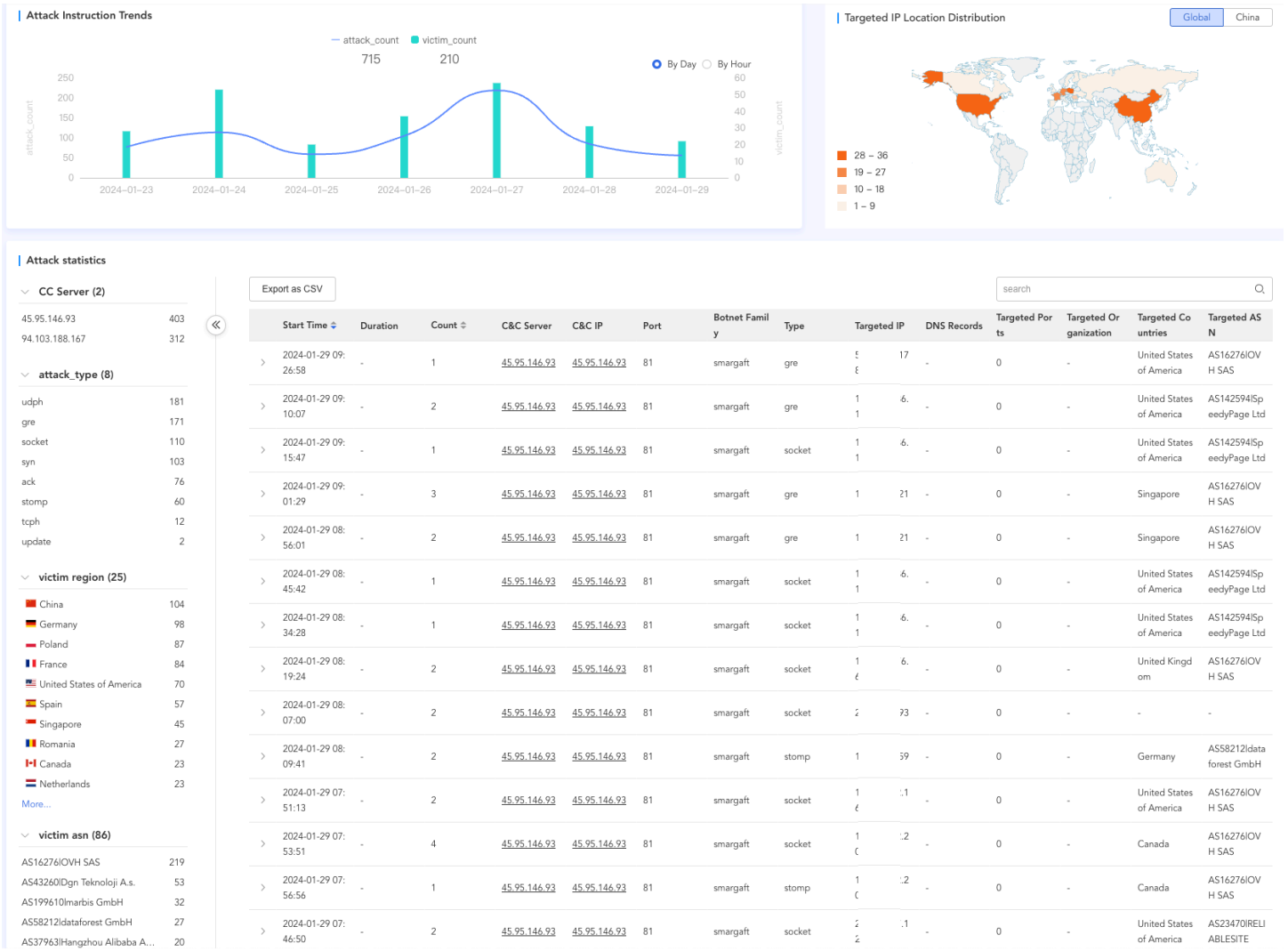
Q: What is eth_call?

Attentive readers will have noticed a Transaction Fee in the "SET C2" diagram, indicating that invoking the method **0x61695f0a** costs money. In fact, calling functions of a smart contract usually incurs a fee. This is not an issue in the "Set C2" scenario, as it's used infrequently; however, in the "Get C2" scenario, a problem arises: "If the number of bots is extremely large, wouldn't each retrieval of C2 information cost a significant amount of money?" This would obviously be unacceptable. Fortunately, Binance's SDK provides a method called **eth_call**, which allows users to read data from a smart contract without generating any blockchain transactions. Since it doesn't alter the state of the blockchain, it doesn't require any fees.

> *XLab notes:*
>
> > ***eth_call*** *was originally designed for simulating contract execution to read data or for testing purposes, without generating any actual effects; it doesn't even get recorded on the blockchain. Therefore, you can retrieve your data (malicious payload) for free, without leaving any traces, and in a robust manner, without leaving behind any evidence.*

DDoS Attack Statistics

Looking at the geographical locations of the targets, the Smargaft botnet attacks are global and not targeted at any specific region. The main areas affected include China, Poland, the USA, Germany, and France. The specific statistics are shown in the following chart.



An interesting observation is that the attack targets of Smargaft are simultaneously assaulted by several different botnets. This leads us to speculate that there is a DDoS platform that aggregates many different botnets.

Sample Propagation Analysis

Based on our data, Smargaft spreads by exploiting known vulnerabilities to propagate a Downloader onto target devices. Once the Downloader is successfully implanted, it then downloads the Bot sample to spread further.

The vulnerabilities utilized by Smargaft include:

VULNERABILITY	AFFECTED
CVE-2013-5948	ASUS RT-AC68U other RT series routers
CVE-2020-8515	DrayTek Vigor Router
LILIN_DVR_RCE	LILIN DVR
TVT_API_RCE	TVT DVR

Downloader Analysis

For our analysis, we focus on the Downloader designed for MIPS architecture. Its basic information is as follows, with the sample using a standard UPX packer for protection.

```
MD5: 2cf03e7425da7244be659d53a972708c
Magic: ELF 32-bit LSB executable, MIPS, MIPS-I version 1 (SYSV), statically linked,
Packer: UPX
```

The core functionalities of the Downloader are straightforward and primarily cover two aspects:

1. **Download the next stage bot sample**: The primary purpose of the Downloader is to fetch and run the next-stage Bot sample.
2. **Eliminate competitors**: Additionally, the Downloader aims to remove any competing malware that may already be present on the device. This ensures that Smargaft secures its control over the device by clearing out potential threats from other malicious actors.

Download & Execute Bot

When the Downloader wants to get the Bot sample, it connects to a download server using port 82. It sends a request to this server, telling it what kind of architecture it's on (like saying, "Hey, I need a MIPS version"). The server, after

getting this request, first sends back a short message, a 10-byte string, that tells how big the Bot file is going to be. After that, it starts sending over the Bot sample.

In a real-world exchange, you'd see the Downloader asking to download a MIPS version of the Bot, with the Bot file being 915,732 bytes big.

Eliminate Competitors

The Downloader uses the /proc filesystem to monitor system processes and forcibly terminates specific ones to take exclusive control of the device.

1. 4 File Transfer Process

```
curl
wget
tftp
ftpget
```

2. 112 System Process

```
init
[kthreadd]
[ksoftirqd/0]
[khelper]
...
total 112 system process name
```

The code below helps figure out if a system process is real or fake by looking at two main things: first, it checks if the process's executable file can be opened and used normally; second, it looks at when that file was last changed. By doing this, we can tell the difference between actual system processes and those that might be pretending to be real or are from competitors.

Bot Analysis

We captured Smargaft bot samples for three different CPU architectures: ARM, MIPS, and X86/64, belonging to two distinct versions. The primary difference between these versions lies in their ability to propagate like a worm. This article focuses on analyzing an older version of the X64 sample. Its basic information is as follows, with the sample utilizing standard UPX packing.

```
MD5: 7f741495f14c828c20db4de6251673fd
Magic: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), statically linked, corr
Packer: UPX
Version: V0
```

Smargaft operates relatively simply. When it runs on a device that has been compromised, it first checks the current user; if it's root, it initiates additional scanning and propagation tasks. Then, it binds to a local port to ensure only one instance runs at a time and manipulates the watchdog to prevent the device from rebooting. Next, it initializes five tasks, including obtaining C2 through smart contracts, conducting DDoS attacks, and ensuring its persistence on the device.Finally, Smargaft runs in an infinite loop, cycling through these tasks at preset intervals. The following discussion will dissect the implementation of Smargaft's functionalities centered around these tasks.

1. Propagation Task
2. GetG2 Task
3. DDoS Task
4. Forward Task
5. Persistence Task
6. Killer Task

Propagation Task

Smargaft spreads by scanning for open ports on devices within the 10.0.0.0/8 IP range, targeting 22 specific ports. It counts the number of IPs checked with a variable `v4`. At `v4=100`, it tries to infect devices with open ports using the

CVE_2021_41653 vulnerability. When `v4 reaches 500`, it switches to randomly scanning public IPs.

There's a bug in the code that prevents Smargaft from scanning public networks as intended: `v4` gets reset to 0 after reaching 100, so the code for `v4=500` never runs, losing the ability for wider network scanning. In Smargaft's newer version, this wasn't fixed; the feature was simply removed.

Regarding the payload constructed for CVE_2021_45653 by Smargaft:

When crafting the SYN packet, Smargaft uses a source port of 55555. The actual sniffing traffic clearly shows a pattern of **1 IP: 22 Port**, which aligns with the analysis mentioned above.

Common Tasks

0x00: Initialization

Smargaft initializes tasks using a specific code snippet and inserts them into a task chain.

Through analysis, the structure of a Task is identified, containing information such as **task method, last task start time, task interval, task type, and next task**.

```
struct Task
{
  _QWORD *task_proc;
  _QWORD last_time;
  _DWORD interval_time;
  _DWORD task_type;
  Task *task_next;
};
```

After redefining relevant variables in IDA as Task* type, the clarity of the task structure is significantly improved.

Smargaft supports five different tasks, each with its detailed attributes.

TYPE	TASK	INTERVAL(SECOND)
1	GetC2	3000
2	DDoS	8
3	Ip-Forward	360000
4	Persistence	360000
5	Killer	1

After all tasks are initialized, they are cycled through in an infinite loop, ensuring each task is executed according to its scheduled interval and type.

0x01: Get C2

This task runs every 3000 seconds and begins by constructing a JSON object using the following code snippet.

The generated JSON appears as follows:

```
{
jsonrpc: "2.0",
method: "eth_call",
id: 1,
params: [
{
to: "0xdf2208d4902aa1ec9a0957132ca86a4e1d40455b",
data: "0xd7ec3ad7"
},
"latest"
]
}
```

This JSON object is for an RPC (Remote Procedure Call) request to the Binance Smart Chain, with each field meaning:

1. `jsonrpc`: The JSON RPC protocol version, here "2.0". JSON RPC is a lightweight protocol for remote procedure calls, allowing requests with

specific commands to Ethereum nodes.

2. `method`: The method being called, here "eth_call". The `eth_call` method executes a function call of a smart contract without causing any state change on the blockchain.
3. `id`: A unique identifier for the request, here 1. This ID distinguishes between different requests and responses, usually a number or string.
4. `params`: An array containing the parameters needed for the method.
    - The first parameter is an object with two fields:
      a. `to`: The address of the smart contract being called, here "0xdf2208d4902aa1ec9a0957132ca86a4e1d40455b".
      b. `data`: Encoded data for the contract function call, here "0xd7ec3ad7", usually a hash of a specific function signature.
    - The second parameter specifies the block state, here "latest" is used to execute the call with the state of the latest block.

> *XLab notes:*
>
> *"latest" always retrieves the most current state in the block, offering a convenient way for automatic C2 updates. Imagine a scenario where all C2s in the IOC database are blocked. The author only needs to update the C2 via the contract, and the Bot, without any updates and leveraging the "latest" feature, can resend the RPC request to access the new C2.*

After generating the JSON, Smargaft randomly selects one from 14 hardcoded RPC nodes to send the request and parses the 'result' value from the returned JSON.

The equivalent script is shown below:

```
curl -X POST URL -d '{ "jsonrpc": "2.0", "method": "eth_call", "id": 1, "params": [
```

Using `https://rpc.ankr.com/bsc` as an example, replace the URL in the script mentioned above with `https://rpc.ankr.com/bsc`. Running it afterward will directly get the following `result` value.

```
0x0000000000000000000000000000000000000000000000000000000000000020
0000000000000000000000000000000000000000000000000000000000000002b3
4352e39352e3134362e39333b39342e3130332e3138382e3136373b3138352e313
3322e3132352e31393300000000000000000000000000000000000000000000
```

To read the C2 from the `result`, start at the 122nd byte and continue until hitting a non-zero digit. Then, decode the following data as a hex string, where the first byte indicates the C2 list's length and the rest is the C2 list, separated by `;`.

```
00000000  2b 34 35 2e 39 35 2e 31 34 36 2e 39 33 3b 39 34  |+45.95.146.93;94|
00000010  2e 31 30 33 2e 31 38 38 2e 31 36 37 3b 31 38 35  |.103.188.167;185|
00000020  2e 31 33 32 2e 31 32 35 2e 31 39 33              |.132.125.193|
```

The Bot connects to the C2 on port 81, sends a `ready\x00` packet, and checks if the response is 0 or more bytes to confirm C2 is live. The first live C2 found is used for DDoS tasks.

0x02: DDoS Task

This task runs every 8 seconds. It establishes a connection with the C2 on port 81, sends a 5-byte "ready" packet, and receives commands from the C2. It supports executing system commands, conducting DDoS attacks, and providing socks5 proxy functionality.

By cross-referencing the variable `v205`, it's evident that the communication protocol is text-based, and a total of 15 different commands are supported.

Below are the commands and their respective functionalities：

CMD	FUNTION
ack	DDoS Vector
syn	DDoS Vector
gre	DDoS Vector
tcph	DDoS Vector
udpg	DDoS Vector
udph	DDoS Vector
httph	DDoS Vector
stomp	DDoS Vector
spoof_vse	DDoS Vector
spoof_syn	DDoS Vector
socket	DDoS Vector
socks	socks5 proxy
kill	kill self
exec	exec system cmds
update	bot update

Using a real captured attack as an example:

When the Bot receives the command, it launches a DDoS attack on port 17481 of the IP 43.249.192.173 using the UDPH attack vector.

0x03: IP-Forward Task

This task runs every 100 hours, first enabling packet forwarding by setting `/proc/sys/net/ipv4/ip_forward` to 1.

Then, the Bot sends a randomly generated 256-byte UDP packet to C2's port 8083. If running with root permissions, it also constructs a UDP packet with a source address of 1.1.1.1 and sends it to C2's port 8083.

0x04: Persistence Task

This task runs every 100 hours with two main objectives:

1. It uses the `Interfere_proc` function to mount one of `tmpfs`, `devpts`, `minix`, or `sysfs` onto the `/proc/pid` directory associated with the bot, preventing tools that rely on the /proc filesystem from obtaining accurate information about the bot process.

   Taking "netstat" as an example, the effect is that it can no longer correctly display information like the process's PID and Program Name.

2. The `Infect_sh` function implements a virus-like infection for files with the .sh suffix in a specified directory by appending \n\n\n bot absolute path &\n to their end. This ensures the bot gets executed every time such a script runs. The function's first parameter is the directory, and the second is the maximum number of layers for infection.

   To demonstrate the infection effect, we created a "goat" directory under the /home directory.

   ```
   ./goat/
   ├── 1.sh
   └── 3layer
       ├── 2.sh
       └── 4layer
           └── 3.sh
   ```

   In practice, the infection of the "goat" and "/etc" directories is as follows: many `.sh` files have had the execution statement `/home/kali/sample/main.x86.unp &` added to their ends. The file `/home/goat/3layer/4layer/3.sh` was not infected because its directory depth exceeded 3.

0x05: Killer Task

This task runs every second, checking each process's command line via `/proc/[PID]/cmdline`. If it contains "-sh" or "tftp", the process is immediately terminated.

The actual effect is as follows, you can see the tftp process being terminated.

Summary

Smargaft's use of EtherHiding for "C2 hosting" showcases significant advantages by harnessing blockchain's decentralization, transparency, and immutability. This method, immune to interventions at the blockchain level, represents an advanced "bulletproof" hosting technique. With smart contracts being relatively easy to learn, it's expected that more malware creators will turn to this technology. The new era of threats is upon us—Stay Vigilant, Stay Safe.

Contact Us

If you're intrigued by our research, reach out to us on Twitter. Also, we have a favor to ask: our DDoS data suggests coordinated attacks by various botnets, yet we're not fully versed in the DDoS underground. If you have insights, we'd love to hear from you.

IOC

sample md5

```
version:v0
```

```
ab794804d7ff5b60327c5051281af80d
e1967081d11debe9757b96a5bb350fa6
b9a9ce54a3b695adcafa9afa556ffac9
38f441481e8765112fb83e556fb33076
7b8c1aa92861e24c81376c6ccb620da5
78bfeff2b2303b9fa51b087c4d762687
7f741495f14c828c20db4de6251673fd

version: v1

8171c8238bd465772b70b943305279cd
323658f0151ec7b7009523e2c368963b
ae64e92bb05ab7ece41f45c79d8e3e6e
88996cbc2658d1294e55863a9fefafaf
dc9cdae709ce4fdf1ceb8b70dd108d36
```

C2(port:81)

```
45.95.146.93    The Netherlands|Noord-Holland|Amsterdam AS49870|Alsycon B.V.
94.103.188.167  Moldova|None|None       AS200019|ALEXHOST SRL
185.132.125.193 China|Hongkong|Hongkong AS9009|M247 Europe SRL
```

Downloader(port:82)

```
45.95.146.93    The Netherlands|Noord-Holland|Amsterdam AS49870|Alsycon B.V.
94.103.188.167  Moldova|None|None       AS200019|ALEXHOST SRL
185.132.125.193 China|Hongkong|Hongkong AS9009|M247 Europe SRL
```

What do you think?

0 Responses

 Upvote

 Funny

 Love

 Surprised

 Angry

 Sad

0 Comments Login ▾

G Start the discussion...

LOG IN WITH OR SIGN UP WITH DISQUS

Name

♡ Share Best Newest Oldest