eBPF Covert Channel Rootkit

Table of Contents

1. Introduction

1.1. Project Description

A covert channel rootkit is a program that masks its existence and enables access and transfer of information in a manner that violates the system's security policy. Rootkits usually provide an attacker with command and control capabilities over the infected host, for example, by enabling remote command execution.

The developed kernel mode covert channel rootkit utilizes eBPF technology. eBPF technology allows the safe extension of capabilities of the kernel, and provides a stable API that will not break between different kernel versions. The developed rootkit has two main components: a client program named Nyatto, and a server program named Nyako.

The implemented application supports transferring of data secretly using HTTP If-None-Match header for concealing messages. The rootkit was designed with the goal to make it hard to detect on the infected host. Stealthiness is achieved by hiding the rootkit’s process from the /proc/ directory, and the output of any linux commands that reveal the rootkit’s pid, such as *ps*.

1.1.1. Essential Problems

Rootkit developers aim to develop a tool that:

1. Provides useful ways to exploit the infected host
2. Can operate without being detected (hide its presence and does not crash the infected host)
3. Is platform version independent and can be used across different kernel versions

Most of the open source rootkits are able to achieve the first goal, however achieving the second and third goals has proven to be challenging using a common approach of implementing rootkit’s modules as Loadable Kernel Modules (LKMs) as they suffer from multiple limitations. Firstly, logical errors in kernel modules risk corrupting the kernel and crushing the system due to lack of security boundaries, which increases the chance of rootkit being detected on the system. Secondly, rootkits that are developed as LKMs can only target a limited number of kernel versions, as their functionality relies on internal memory layout, which can change between kernel releases; the issue can be addressed by re-implementing the same functionality for different kernel releases and configuring rookit’s functionality using pragmas, which introduces additional complexity to the project. Lastly, rootkits developed as kernel modules are hard to develop and debug, as they risk corrupting and crushing the host.

To address the above limitations, this project develops a covert channel rootkit using eBPF technology. eBPF extends the capabilities of the Berkeley Packet Filter, originally developed by Steven McCanne and Van Jacobson. eBPF was implemented as a lightweight virtual machine that can run sandboxed programs in an operating system kernel. eBPF programs address limitations of LKMs by:

- Passing a verification step that ensures they are safe to run
- Making function calls into helper functions, a stable API offered by the kernel

1.1.2. Goals and Objectives

The main goal of this project is to explore usage of eBPF as an alternative technology for developing kernel-mode rootkits by implementing a non-trivial covert channel rootkit. The developed proof-of-concept tool relies on eBPF for implementing the majority of its functionality, such as processing of network packets, hiding itself, and modifying the infected host's behavior. The developed covert channel rootkit has to enable covert exfiltration of information from the target host, hide its presence on the target host, and exchange the data with the remote server without being detected by Intrusion Detection Systems.

2. Body

2.1. Background

Berkeley packet filter (BPF) started as a simple language for writing packet-filtering code. BPF was mostly used for creating utilities like tcpdump. However, over the years the BPF subsystem became more sophisticated, and adapted for solving a wider range of problems by gaining the ability to run sandboxed programs inside the operating system kernel. Extended BPF (eBPF) can be used to safely and efficiently extend the capabilities of the kernel, without requiring to change the kernel source code, or load kernel modules. eBPF solutions are rapidly gaining in popularity in the areas of networking, security, and observability.

Recently, security researchers were able to successfully use eBPF for development of malicious tools; at BlackHat USA 2021, Datadog employees presented a fairly complex rootkit implementation (Fournier, Afchain, Baubeau, 2021); at DEF CON 29, Pat H (pathtofile) successfully presented a set of malicious tools developed using eBPF that demonstrate various offensive techniques (Pat H. 2021).

eBPF can be a powerful tool for developing covert channel rootkits, as it allows for safe modification of the system's behavior using a stable API provided by the kernel. Additionally, because eBPF is a relatively new technology, monitoring tools that can detect malicious eBPF usage are not very common, making the eBPF subsystem an effective attack vector.

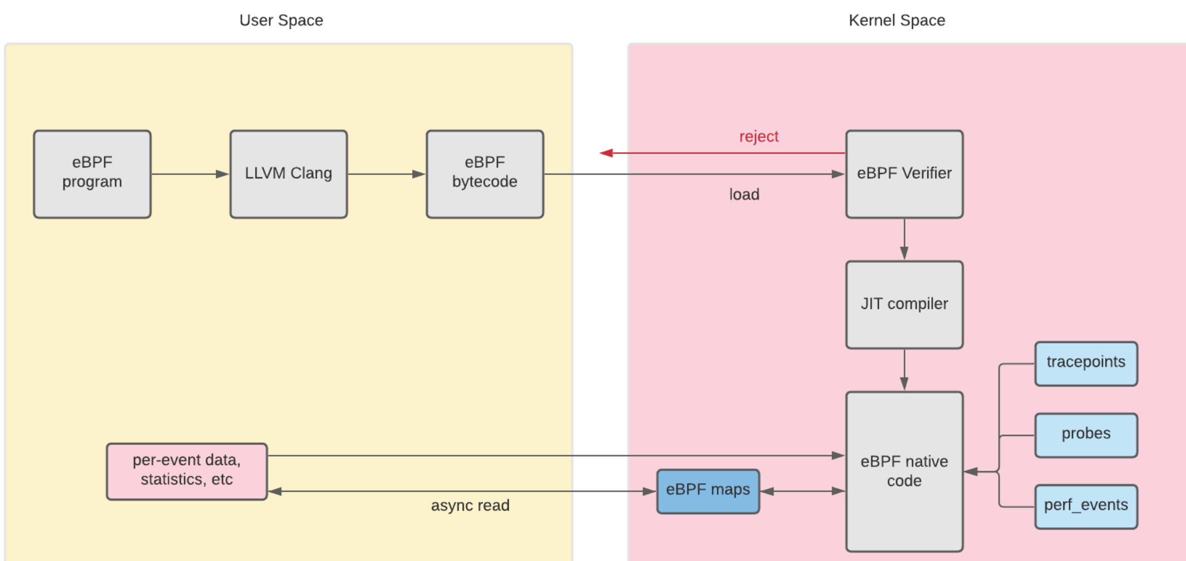

Figure 1: eBPF program overview.

2.2. Design

2.2.1. Implementation Details

2.2.1.1. Overview

The implemented rootkit consists of two components:

- Nyako - server component running on the exploited host
- Nyatta - client component acting as a command and control center

Nyako and Nyatta communicate by exchanging HTTP GET requests with encrypted messages embedded inside If-None-Match Header. The details of the covert channel implementation and the communication protocol are presented in the sections below.

Nyatta performs two functions, the main thread accepts user input, crafts the message and sends it to the server. Another child thread listens for packets from the server containing command results using libpcap.

Nyako consists of multiple modules that communicate between user and kernel space. Kernel space modules are implemented using eBPF; they are responsible for the core rootkits functionality. Every kernel space module has its userspace component that is responsible for polling data from kernel space through various data structures provided by eBPF. Currently Nyako has 3 eBPF modules:

- nyako_kern - implements kernel component of the covert channel
  - Responsible for reading packets as they reach the network interface and making data available to the user space through BPF_MAP_TYPE_ARRAY data structure
  - Implemented in nyako_kern.c file
- pidhide_kern - implements rootkit's pid hiding functionality
  - Hooks into getdents64 system calls and makes them skip the rootkit's linux_dirent64 structure
  - Implemented in pidhide_kern.c file
- no_trace - implements system wide trace blocking
  - Hooks into ptrace system call and sends SIGKILL to any program trying to use it, for example strace
  - Implemented in no_trace_kern.c file

Additional implementation details on the eBPF modules are available in the sections below.

2.2.1.1.1. Configuration

The configuration for Nyako and Nyatta is available inside the corresponding config.h files.

```
#define BACKDOOR_URL "192.168.56.11"
#define BACKDOOR_PORT 80
#define LISTEN_PORT 80
#define DEBUG_ENABLED 0
#define LOCAL false
#define DEV "eth1"
```

Figure 3: configuration values for Nyatta.

```
#define CLIENT_PORT 80
#define DEBUG_ENABLED 0
```

Figure 4: configuration values for Nyako.

2.2.1.1.2. Debugging

Additional debugging information printed to stdout can be enabled by setting DEBUG_ENABLED to 1. Debugging should only be enabled during development as it generates a lot more I/O operations.

2.2.1.2. System/Software Architecture Diagram

The two finite state diagrams below, for Nyako and Nyatta, visualize the details presented in the Pseudocode section 2.5.2.3.

2.2.1.2.1. Nyako

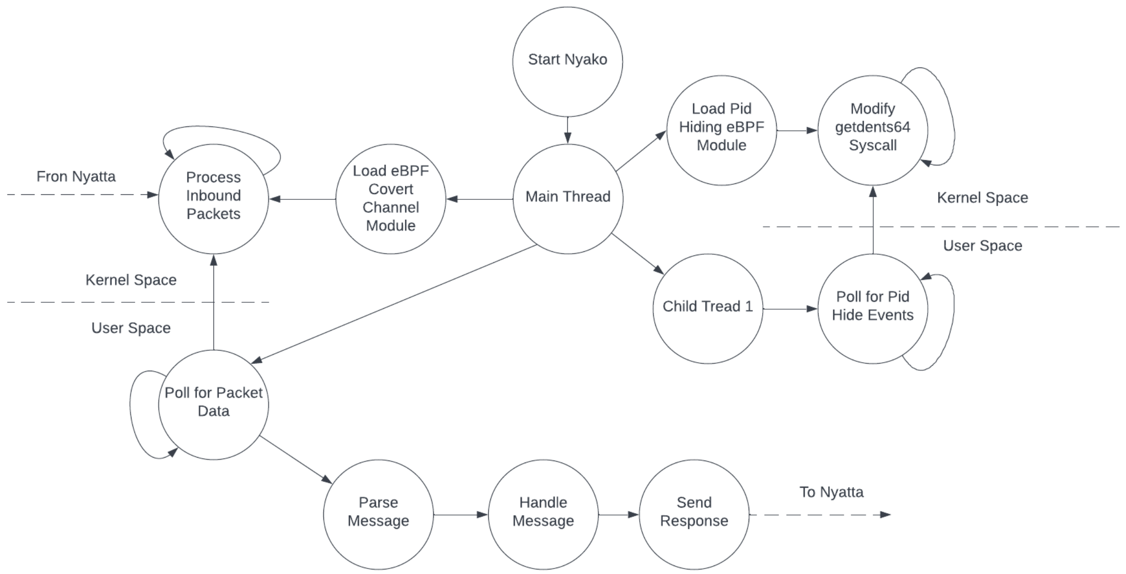

Figure 5: architecture diagram of Nyako.

2.2.1.2.2. Nyatta

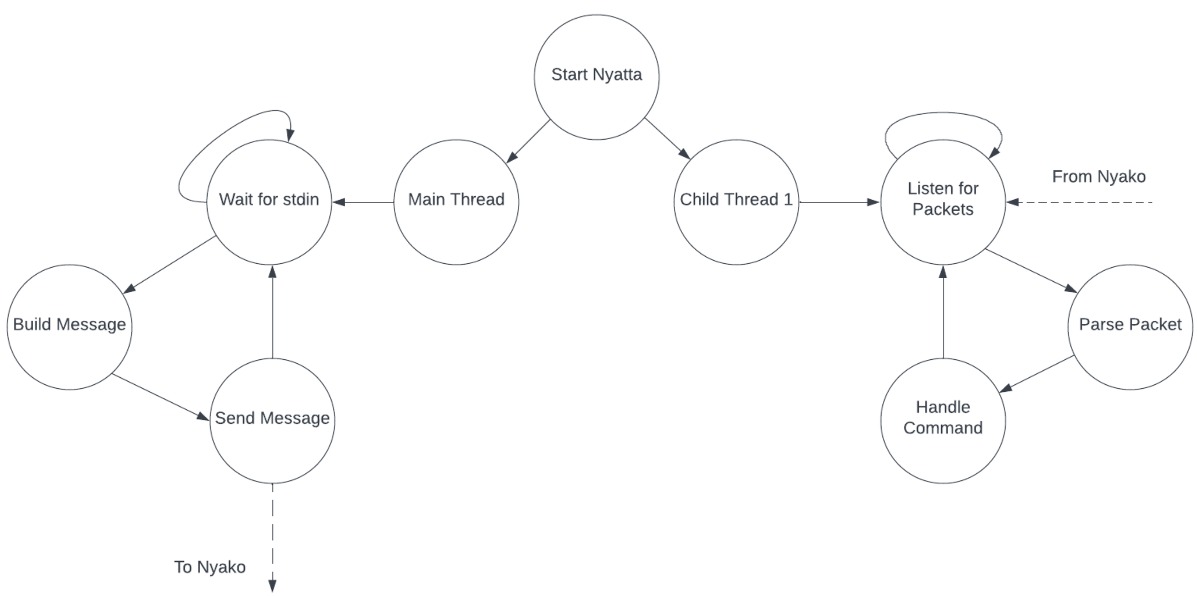

Figure 6: architecture diagram of Nyatta.

2.2.1.3. Pseudocode

2.2.1.3.1. Nyako

```
Function for processing inbound packets

{

      Create BPF map array for storing command queue

      Create map for storing message cout

      Parse packet headers

      Write client's ip to the message details

      Write message from If-None-Match header to the message
      details

      Enqueue message to the queue

}

Function for hiding the rootkit's pid

{

      Loop over linux_dirent64 structs

      {

            if directory name is equal to pid

            {

                  Hide target by changing d_reclen of the
                  previous linux_dirent64 struct to cover the
                  target


                  Write event details to the ring buffer

            }

      }

}

Function for loading eBPF module for hiding the rootkit's
pid

{
```

```
        Load eBPF modules for handing various getdents64
        system calls

        Setup ring buffer for collecting event details

        Every 100 milliseconds poll data from the ring buffer

        {

            If debug mode enabled

            {

                Print results to the terminal

            }

        }

}


Function for loading eBPF module for trace blocking

{

        Load eBPF to block ptrace system call

        Setup ring buffer for collecting event details

        Every 100 milliseconds poll data from the ring buffer

        {

            Print results to the terminal

        }

}


Function hook for ptrace system call

{

        Send SIGKILL to the process

        Write result to the ring buffer

}


Function for handling messages
```

```
{
    Parse message
    If message rootkit is not active and not message type
    is TYPE_INVOKE or TYPE_INVOKE
    {
        Return from function
    }
    If message type is TYPE_EXECUTE_CMD
    {
        Decrypt message
        Execute command
        Collect command results
        Split command result in chunks
        For chunk in chunks
        {
            Encrypt chunk
            Craft message
            Send HTTP GET request with embedded message
        }
    }
    Else if message type is TYPE_INVOKE
    {
        Set rootkit to active
    }
    Else if message type is TYPE_SUSPEND
    {
        Disable offensive eBPF modules
        Set rootkit to inactive
    }
```

```
        Else if message type is TYPE_TERMINATE
        {
              Unload all eBPF modules
              Exit
        }
        Else if message type is TYPE_BLOCK_TRACE
        {
              Create thread for loading block trace eBPF module
        }
        Else if message type is UNBLOCK_TRACE
        {
              Unload eBPF module for system wide trace blocking
        }
        Else
        {
              Print unsupported command error
        }
}

Main Function
{
        Load covert channel eBPF module
        Create thread for loading eBPF module for hiding the
        rootkit's pid
        Every 2 seconds poll for messages from the BPF map
        containing queue of messages
        {
              For every element in the queue
              {
```

```
            Handle message
        }
    }
}
```

2.2.1.3.2. Nyatta

```
Function to receive command results
{
    Parse packet
    Extract data from the If-None-Match header of the HTTP
    request
    Parse message
    If authentication header is present and command type
    is TYPE_SEND_CMD_RESULT
    {
        Decrypt the command result
        Write the result to stdout
    }
}

Main Function
{
    Craft message containing invoke command
    Send HTTP GET request with embedded message
    Create a thread that collects remote command execution
    results
    Loop indefinitely and capture commands to send to the
    server
    {
        Get command type
        Encrypt command
```

```
                Craft message

                Embed message inside HTTP “If-None-Match” header
                field

                Send HTTP GET request with embedded message

        }

    }
```

2.2.1.4. Covert Channel Design

The covert channel is implemented using HTTP protocol, an application layer protocol that is sent over TCP. Both client and server rootkit component messages are embedded inside the If-None-Match header that is sent as part of the GET requests sent to, and from the exploited host. The If-None-Match header was chosen for transferring encrypted data covertly for the following reasons:

- Its value is used to implement resource caching, and it’s hard to detect if the header is dysfunctional as it only results in a slight increase of bandwidth
- The maximum length of an etag_value is roughly 8192 bytes, so it’s possible to covertly transfer large amounts of data
- The etag_value is a randomly generated ASCII string, hand crafted etag values are hard to detect

```
If-None-Match:
lo7ct.0.0.24.nwlrbbmqbhcdarzowkkyhid.nwlrbbmqbhcdarzowkkyhid.dqscdxrjmowfrx
sjybldbefsarcbynecdyggxxpklorellnmpapqfwkhopkmcoqhnwnkuewhsqmgbbuqcljjivswm
dkqtbxixmvtrrbljptnsnfwzqfjmafadrrwsofsbcnuvqhffbsaqxwpqcacehchzvfrkmlnozjk
pqpxrjxkitzyxacbhhkicqcoendtomfgdwdwfcgpxiqvkuytdlcgdewhtaciohordtqkvwcsgsp
qoqmsboaguwnnyqxnzlgdgwpbtrwblnsadeuguumoqcdrubetokyxhoachwdvmxxrdryxlmndqt
ukwagmlejuukwcibxubumenmeyatdrmydiajxloghiqfmzhlvihjouvsuyoypayulyeimuotehz
riicfskpggkbbipzzrzucxamludfykgruowzgiooobppleqlwphapjnadqhdc
```

Figure 7: If-None-Match HTTP header containing encrypted message.

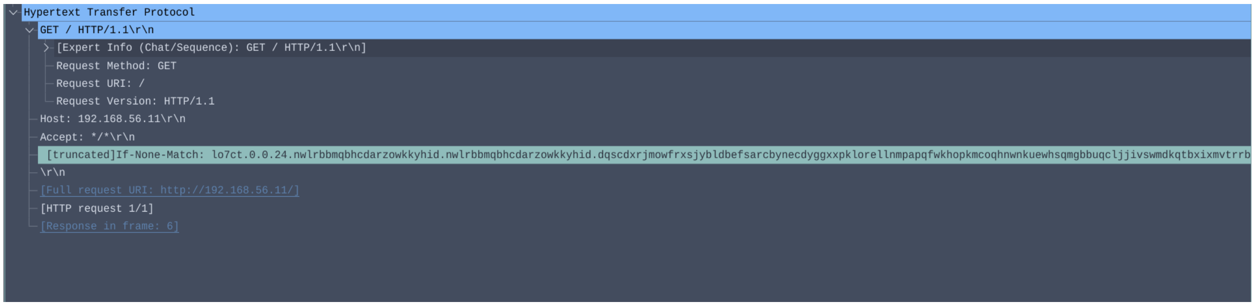

Figure 8: HTTP frame.

The client and server rootkit components exchange data by sending HTTP GET requests to destination port 80. The requests are crafted using libcurl *curl_easy_perform* API.

HTTP GET requests were chosen as the communication medium, because they closely mimic normal traffic, and cannot be detected by Intrusion Detection Systems.

The reliability of the communication is achieved by relying on the features of the TCP protocol.

2.2.1.5. Communication Protocol

2.2.1.5.1. Authentication

Packets related to the rootkit's communication are authenticated by checking AUTH_HEADER constant value at the expected location of the message. The constant is defined in nyako/src/constants.h, and nyatta/src/constants.h:

#define AUTH_HEADER (unsigned char*)"lo7ct"

2.2.1.5.2. Encryption

The data involved in the communication is encrypted using public-key cryptography implemented by libsodium. The generated bytes of ciphertext and nonce are then converted to hexadecimal string to ensure the If-None-Match header value (etag_value) consists only of ASCII characters, as required by the specification.

The keypair can be generated using a script available under utils/generate_keypair.c, the automated key exchange is out of scope of this project (the keys have to be exchanged manually).

2.2.1.5.3. Command Types

The supported command types are defined inside the command_types enumerator.

Command Type	Value	Description
TYPE_INVOKE	0	Command type for rootkit invocation.
TYPE_EXECUTE_CMD	1	Command type indicating remote command execution.
TYPE_SEND_CMD_RESULT	2	Command type indicating the message commands command execution data.
TYPE_SUSPEND	3	Command type for rootkit suspension.
TYPE_BLOCK_TRACE	4	Command type to load eBPF module for blocking ptrace system call.
TYPE_UNBLOCK_TRACE	5	Command type to unload eBPF module for blocking ptrace system call.
TYPE_TERMINATE	6	Command type to terminate rootkit.

2.2.1.5.4. Message Format

Messages exchanged by the rootkit components consist of multiple sections separated by a dot.

Section	Description	Example
Authentication Header	Authentication Header	lo7ct
Type	Numeric command type.	0
ID	Unique message identifier, can be used to ensure the reliability of the communication.	123
Ciphertext Length	Length of encrypted data, necessary for decryption.	256
Ciphertext	Encrypted data.	ludfykgruoapjnadqhdc
Nonce	Randomly generated nonce.	wzgiooobppleqlwphsjf

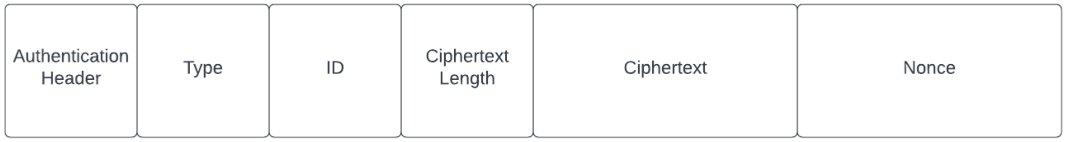

Figure 9: message format diagram.

2.2.1.6. eBPF Modules

The following section provides implementation details on the rootkit components that run in kernel space, and were implemented using eBPF.

2.2.1.6.1. Packet Processing

The packet processing module processes inbound network packets just as they hit the network interface. It's implemented using the eXpress Data Path (XDP) framework, which operates as a hook placed in the network interface controller driver before any memory allocation needed by the network stack itself to achieve high performance. The network packets can be manipulated as bytes, as long as they path the eBPF verifier that ensures the program is safe to run.

The kernel component of the covert channel performs 3 tasks:

1. Parses the packet headers to collect sender's IP and the message embedded inside the If-None-Match header
2. Writes the collected header to the *message_details struct*, defined as following:

```
struct message_details
{
    unsigned char message[MESSAGE_BUF_SIZE];
    unsigned int ip_saddr;
};
```

3. Enqueues the collected message_details to the BPF_MAP_TYPE_ARRAY data structure, which operates as a circular queue

The queue is processed on every poll action from the user space, and was implemented to improve performance, and eliminate the need for frequent polling of data from the kernel space.

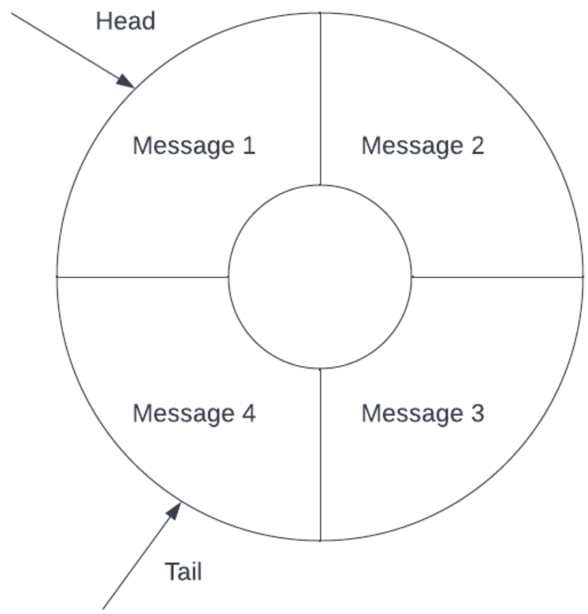

Figure 10: message queue diagram.

2.2.1.6.2. Pid Hiding

The eBPF module responsible for pid hiding of the rootkit is the most complex. It's built upon techniques described by Harvey Phillips in his article. The article outlines how to implement pid hiding as a loadable kernel module (Phillips, 2020).

The basic idea behind PID hiding involves hooking into various points in the lifecycle of the *getdents* system call, which is responsible for directory listing, and modifying the buffer it returns to the user space.

The *linux_dirent64* structure is defined as following:

```
struct linux_dirent64 {
```

```
        u64         d_ino;
        s64         d_off;
        unsigned short      d_reclen;
        unsigned char       d_type;
        char        d_name[];
};
```

The *linux_dirent64* structure contains two fields that are particularly important:

- d_reclen - the total size of the struct in bytes, which is necessary to iterate through the dirent64 structures in memory
- d_name - the name of the directory, which allows searching for the directory structure with the name of the supplied pid

To observe response from the kernel, we need to create an eBPF hook for the exit to the getdents64 syscall (tp/syscalls/sys_exit_getdents64). The next step is to iterate over *dirent64* structures to find the one with *d_name* matching the rootkit's pid. Then, to trick the system into skipping over the entry matching the rootkit's pid, the d_reclen field of the previous entry is incremented to cover itself and our target.

Lastly, the event results are saved to a BPF map for the userspace to poll the event details.

When the patched buffer is returned to the userspace, and a user-space utility (like *ls*) is looping through the entries by reading d_reclen, it will jump over the hidden entry.

2.2.1.6.3. System Call Blocking

The implementation of eBPF module responsible for *ptrace* system call blocking is very simple due to existence of entry to the ptrace syscall endpoint *tp/syscalls/sys_enter_ptrace*. When a ptrace system call is invoked, the module sends a SIGKILL using *bpf_send_signal* helper. Lastly, the event results are saved to a BPF map for the userspace to poll the event details.

2.2.2. User Manual

2.2.2.1. Environment Setup

The environment for performing development and testing was created using Vagrant, a tool for defining software infrastructure as code. The changes to the source code are automatically synchronized between the VMs and the host.

```ruby
Vagrant.configure("2") do |config|
  config.vm.define "nyako" do |server|
    server.vm.box = "fedora/34-cloud-base"
    server.vm.network "private_network", ip: "192.168.56.11"
    server.vm.hostname = "nyako"
    server.vm.define "nyako"
    server.vm.provision :shell, path: "setup_nyako.sh"
    server.vm.synced_folder "nyako", "/home/vagrant/nyako"
    server.vm.provider "virtualbox" do |vb|
      vb.memory = "2048"
      vb.cpus = "2"
    end
  end

  config.vm.define "nyatta" do |client|
    client.vm.box = "fedora/34-cloud-base"
    client.vm.network "private_network", ip: "192.168.56.12"
    client.vm.hostname = "nyatta"
    client.vm.define "nyatta"
    client.vm.provision :shell, path: "setup_nyatta.sh"
    client.vm.synced_folder "nyatta", "/home/vagrant/nyatta"
    client.vm.provider "virtualbox" do |vb|
      vb.memory = "2048"
      vb.cpus = "2"
    end
  end
end
```

Figure 11: Vagrantfile.

The environment consists of two virtual machines with the following characteristics:

- Target Host
  - Operating System: Fedora 34
  - IP address: 192.168.56.11
  - Hostame: “nyako”
- Attacker Host:
  - Operating System: Fedora 34
  - IP address: 192.168.56.12
  - Hostame: “nyatta”

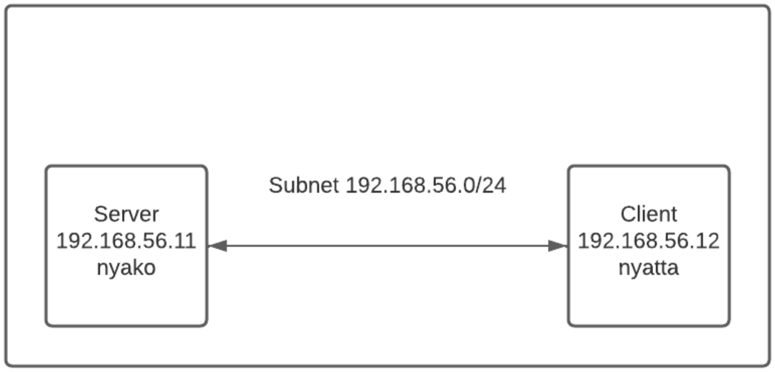

Figure 12: development environment setup.

The virtual machines are defined inside the Vagrantfile located at the root directory. The packages necessary for development and testing are defined inside setup_nyako.sh and setup_nyatta.sh bash scripts.

The environment can be created using the *vagrant up* command. After the commands execution the VMs are accessible via *ssh <hostname>* command.

```
[maksymc@maksym-hpspectrex360convertible13aw0xxx Source]$ vagrant up
Bringing machine 'nyako' up with 'virtualbox' provider...
Bringing machine 'nyatta' up with 'virtualbox' provider...
==> nyako: Importing base box 'fedora/34-cloud-base'...
==> nyako: Matching MAC address for NAT networking...
==> nyako: Checking if box 'fedora/34-cloud-base' version '34.20210423.0' is up to date...
==> nyako: Setting the name of the VM: Source_nyako_1648339159510_92889
==> nyako: Clearing any previously set network interfaces...
==> nyako: Preparing network interfaces based on configuration...
    nyako: Adapter 1: nat
    nyako: Adapter 2: hostonly
==> nyako: Forwarding ports...
    nyako: 22 (guest) => 2222 (host) (adapter 1)
==> nyako: Running 'pre-boot' VM customizations...
==> nyako: Booting VM...
==> nyako: Waiting for machine to boot. This may take a few minutes...
```

Figure 13: environment creation.

2.2.2.2. Compilation

Nyako and Nyatta can be compiled using the available Makefiles.

2.2.2.2.1. Nyatta

To compile Nyatta run: *make nyatta*.

```
[root@nyatta nyatta]# make nyatta
gcc -Wall -o nyatta src/logger.c src/utils.c src/crypto.c src/message
.c src/network.c src/listener.c src/nyatta.c -lpcap -lsodium -pthread
 -lcurl -lnet
```

Figure 14: Nyatta compilation.

2.2.2.2.2. Nyako

To compile Nyako run: *make nyako*.

```
[root@nyako nyako]# make nyako
clang -g -O2 -Wall -target bpf -c src/nyako_kern.c -o nyako_kern.o
clang -g -O2 -Wall -target bpf -c src/no_trace_kern.c -o no_trace_kern.o
bpftool gen skeleton no_trace_kern.o > src/no_trace_skeleton.h
clang -g -O2 -Wall -target bpf -c src/pidhide_kern.c -o pidhide_kern.o
bpftool gen skeleton pidhide_kern.o > src/pidhide_skeleton.h
clang -g -O2 -Wall -o nyako.o src/utils.c src/logger.c src/crypto.c src/message.c sr
c/network.c src/bpf_helpers.c src/no_trace.c src/pidhide.c src/nyako.c -lsodium -lcu
rl -lbpf -pthread
```

Figure 15 Nyako compilation.

2.2.2.3. Usage Instructions

Nyato and Nyatta do not accept any command line arguments, the rootkit configuration can be performed by updating corresponding config.h files. The configuration options are self explanatory. The cheatsheet.txt file contains commonly used helper commands.

```
[root@nyako nyako]# ./nyako.o
hiding PID 3662 ...
```

Figure 16: Nyako execution.

Remote command execution can be performed by entering a linux command. Rootkit specific commands are outlined in the table below.

Command	Description
invoke	Send a message with the command type TYPE_INVOKE.
suspend	Send a message with the command type TYPE_SUSPEND.
block_trace	Send a message with the command type

	TYPE_BLOCK_TRACE.
unblock_trace	Send a message with the command type TYPE_UNBLOCK_TRACE.
terminate	Send a message with the command type TYPE_TERMINATE.

```
[root@nyatta nyatta]# ./nyatta
INFO: created data loop with filter: src host 192.168.56.11 and port 80
terminate
```

Figure 17: Nyatta execution.

3. References

Andrii Nakryiko. (2021) BPF tips & tricks: the guide to bpf_trace_printk() and bpf_printk(). https://nakryiko.com/posts/bpf-tips-printk/

Joanna Rutkowska. (2004). The Implementation of Passive Covert Channels in the Linux Kernel.

http://www.infosecwriters.com/text_resources/pdf/passive_covert_channels_linux.pdf

Wenhua Zhao. (2019) Workaround for a bpf verifier error. https://mechpen.github.io/posts/2019-08-29-bpf-verifier/index.html

Guillaume Fournier, Sylvain Afchain, Sylvain Baubeau. (2021). With Friends Like eBPF, Who Needs Enemies?

https://www.blackhat.com/us-21/briefings/schedule/#with-friends-like-ebpf-who-needs-enemies-23619

Alok Tiagi, Hariharan Ananthakrishnan, Ivan Porto Carrero, Keerti Lakshminarayan. (2021). How Netflix uses eBPF flow logs at scale for network insight.

https://netflixtechblog.com/how-netflix-uses-ebpf-flow-logs-at-scale-for-network-insight-e3ea997dca96

Julia Evans. (2017). ftrace: trace your kernel functions!

https://jvns.ca/blog/2017/03/19/getting-started-with-ftrace/

Hangbin Liu. (2021). tc/BPF and XDP/BPF. https://liuhangbin.netlify.app/post/ebpf-and-xdp/

Pat H. (2021). DEF CON 29: Bad BPF - Warping reality using eBPF.

https://blog.tofile.dev/2021/08/01/bad-bpf.html

Harvey Phillips. (2020). Linux Rootkits Part 6: Hiding Directories.
https://xcellerator.github.io/posts/linux_rootkits_06/