Final Report on Monitoring User Programs with eBPF

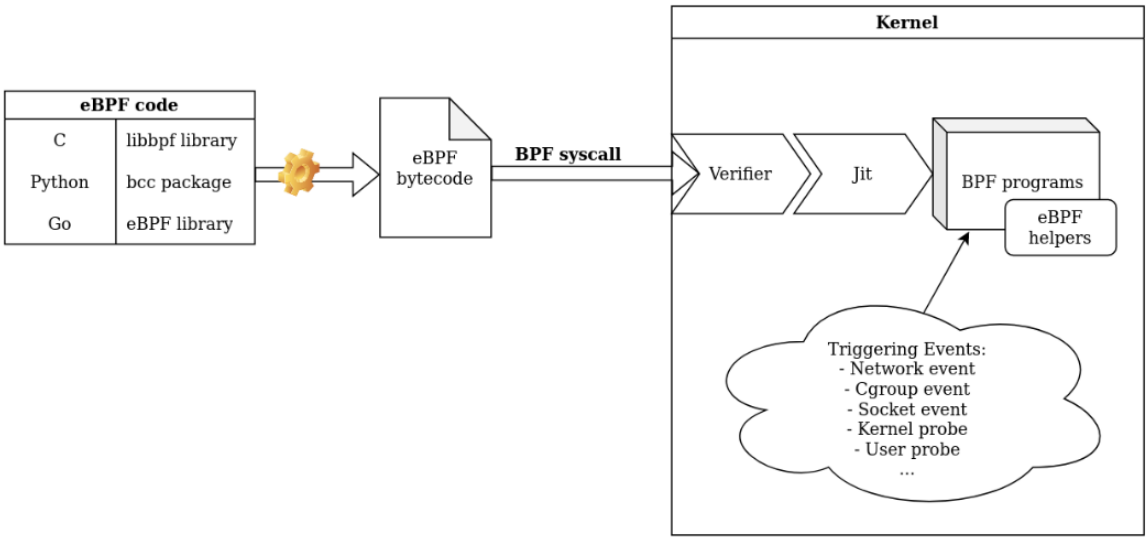
Abstract

This report delves into the application of extended Berkeley Packet Filter (eBPF) for monitoring user programs in Linux environments. eBPF emerges as a potent tool for intercepting and analyzing system operations, offering unparalleled insights into both kernel and user space activities. The focus is primarily on exploiting uprobes (userland probes) for a more detailed scrutiny of user space functions. A visual representation of the eBPF operational framework is provided, elucidating the transition from eBPF C/Python code to kernel execution. The report outlines the process of setting up uprobes and illustrates their utility through a detailed example involving a simple web server implemented in C. The methodology incorporates the BCC toolkit, and the eBPF program is developed in C and Python, focusing on capturing system calls and network traffic. The highlight is a practical demonstration where eBPF is applied to monitor the latency of specific functions within a homebrewed web server. This exercise not only solidifies the understanding of eBPF's capabilities but also showcases its practical implications in real-world system monitoring and performance analysis. The report culminates with an evaluation of the eBPF tool in action, providing empirical data and insights into its efficiency and effectiveness in monitoring user program latencies

Introduction

eBPF (extended Berkeley Packet Filter) is a powerful tool in the Linux environment for observing and tracking system operations. This advanced technology allows for the interception and analysis of various system events, providing deep insights into system behavior. With eBPF, you have the capability to delve into both kernel and userspace functions, opening up possibilities like decrypting and examining communications between processes or tracing the usage of specific functions in libraries. Understanding the mechanics and potential applications of eBPF can be greatly facilitated by considering a visual representation or flow diagram that illustrates its operational framework.

When embarking on the creation of an eBPF application, your first step is to select an appropriate eBPF library. This library is responsible for generating the eBPF bytecode, which is then injected into the kernel using the bpf syscall. Once in the kernel, your eBPF code undergoes a verification process to ensure its safety before it's executed. It's important to note that eBPF programs come in various forms, each tailored to specific types of events. These variants provide access to unique sets of eBPF helpers and contexts, depending on their purpose. In the realm of system monitoring, eBPF is often utilized through kprobes (kernel probes), enabling the tracking of system calls by processes. However, this method doesn't capture all possible data of interest. To address this limitation, ongoing research is exploring uprobes (userland probes) for more comprehensive monitoring capabilities within user space.



The above figure shows the process of a eBPF C/Python code to Kernel execution. First the developers writes eBPF Code with a restricted subset of C or Python, which perform tasks like monitoring system calls, network packets or system behavior analysis. Then the eBPF code can be compiled to eBPF bytecode using LLVM compilers. Then the system loads the bytecode via eBPF syscall into the kernel space. Within the kernel, there will be a eBPF verifier which checks the bytecode for safety, stability and security. After that, a Just-In-Time(JIT) Compiler will be used to convert eBPF bytecode into actual machine code, which guarantees faster execution. Finnaly, the CPU executes the eBPF program.

eBPF uprobes review

Uprobes, a kernel functionality, offer the capability to attach hooks to any instruction within any userland program. When activated, these hooks generate events and supply the context of the targeted program to designated handlers, such as an eBPF program. This enables various actions, like logging the values of CPU registers or executing specific eBPF routines.

Creating a uprobe

Developers can create a uprobe using the /sys pseudo-filesystem by adding a line to the /sys/kernel/debug/tracing/uprobe_events file.

```
p[:[GRP/]EVENT] PATH:OFFSET [FETCHARGS] : Set a uprobe
r[:[GRP/]EVENT] PATH:OFFSET [FETCHARGS] : Set a return uprobe (uretprobe)
-:[GRP/]EVENT                           : Clear uprobe or uretprobe event
```

Based on the Linux Kernel documentation, the usage and the available events are as follows:

```
Overview
--------
Uprobe based trace events are similar to kprobe based trace events.
To enable this feature, build your kernel with CONFIG_UPROBE_EVENTS=y.

Similar to the kprobe-event tracer, this doesn't need to be activated via
current_tracer. Instead of that, add probe points via
/sys/kernel/debug/tracing/uprobe_events, and enable it via
```

```
/sys/kernel/debug/tracing/events/uprobes/<EVENT>/enabled.

However unlike kprobe-event tracer, the uprobe event interface expects the
user to calculate the offset of the probepoint in the object.

Synopsis of uprobe_tracer
-------------------------
  p[:[GRP/]EVENT] PATH:OFFSET [FETCHARGS] : Set a uprobe
  r[:[GRP/]EVENT] PATH:OFFSET [FETCHARGS] : Set a return uprobe (uretprobe)
  -:[GRP/]EVENT                           : Clear uprobe or uretprobe event

  GRP           : Group name. If omitted, "uprobes" is the default value.
  EVENT         : Event name. If omitted, the event name is generated based
                  on PATH+OFFSET.
  PATH          : Path to an executable or a library.
  OFFSET        : Offset where the probe is inserted.

  FETCHARGS     : Arguments. Each probe can have up to 128 args.
   %REG         : Fetch register REG
   @ADDR    : Fetch memory at ADDR (ADDR should be in userspace)
   @+OFFSET : Fetch memory at OFFSET (OFFSET from same file as PATH)
   $stackN  : Fetch Nth entry of stack (N >= 0)
   $stack   : Fetch stack address.
   $retval  : Fetch return value.(*)
   $comm    : Fetch current task comm.
   +|-offs(FETCHARG) : Fetch memory at FETCHARG +|- offs address.(**)
   NAME=FETCHARG     : Set NAME as the argument name of FETCHARG.
   FETCHARG:TYPE     : Set TYPE as the type of FETCHARG. Currently, basic
types
               (u8/u16/u32/u64/s8/s16/s32/s64), hexadecimal types
               (x8/x16/x32/x64), "string" and bitfield are supported.

  (*) only for return probe.
  (**) this is useful for fetching a field of data structures.

Types
-----
Several types are supported for fetch-args. Uprobe tracer will access
memory
by given type. Prefix 's' and 'u' means those types are signed and unsigned
respectively. 'x' prefix implies it is unsigned. Traced arguments are shown
in decimal ('s' and 'u') or hexadecimal ('x'). Without type casting, 'x32'
or 'x64' is used depends on the architecture (e.g. x86-32 uses x32, and
x86-64 uses x64).
String type is a special type, which fetches a "null-terminated" string
from
user space.
Bitfield is another special type, which takes 3 parameters, bit-width, bit-
offset, and container-size (usually 32). The syntax is;

 b<bit-width>@<bit-offset>/<container-size>

For $comm, the default type is "string"; any other type is invalid.


Event Profiling
```

```
---------------
You can check the total number of probe hits and probe miss-hits via
/sys/kernel/debug/tracing/uprobe_profile.
The first column is event name, the second is the number of probe hits,
the third is the number of probe miss-hits.

Usage examples
--------------
 * Add a probe as a new uprobe event, write a new definition to
uprobe_events
as below: (sets a uprobe at an offset of 0x4245c0 in the executable
/bin/bash)

    echo 'p /bin/bash:0x4245c0' > /sys/kernel/debug/tracing/uprobe_events

 * Add a probe as a new uretprobe event:

    echo 'r /bin/bash:0x4245c0' > /sys/kernel/debug/tracing/uprobe_events

 * Unset registered event:

    echo '-:p_bash_0x4245c0' >> /sys/kernel/debug/tracing/uprobe_events

 * Print out the events that are registered:

    cat /sys/kernel/debug/tracing/uprobe_events

 * Clear all events:

    echo > /sys/kernel/debug/tracing/uprobe_events

Following example shows how to dump the instruction pointer and %ax
register
at the probed text address. Probe zfree function in /bin/zsh:

    # cd /sys/kernel/debug/tracing/
    # cat /proc/`pgrep zsh`/maps | grep /bin/zsh | grep r-xp
    00400000-0048a000 r-xp 00000000 08:03 130904 /bin/zsh
    # objdump -T /bin/zsh | grep -w zfree
    0000000000446420 g    DF .text  0000000000000012  Base        zfree

  0x46420 is the offset of zfree in object /bin/zsh that is loaded at
  0x00400000. Hence the command to uprobe would be:

    # echo 'p:zfree_entry /bin/zsh:0x46420 %ip %ax' > uprobe_events

  And the same for the uretprobe would be:

    # echo 'r:zfree_exit /bin/zsh:0x46420 %ip %ax' >> uprobe_events

Please note: User has to explicitly calculate the offset of the probe-point
in the object. We can see the events that are registered by looking at the
uprobe_events file.

    # cat uprobe_events
    p:uprobes/zfree_entry /bin/zsh:0x00046420 arg1=%ip arg2=%ax
```

```
    r:uprobes/zfree_exit /bin/zsh:0x00046420 arg1=%ip arg2=%ax

Format of events can be seen by viewing the file
events/uprobes/zfree_entry/format

    # cat events/uprobes/zfree_entry/format
    name: zfree_entry
    ID: 922
    format:
         field:unsigned short common_type;         offset:0;  size:2;
signed:0;
         field:unsigned char common_flags;         offset:2;  size:1;
signed:0;
         field:unsigned char common_preempt_count; offset:3;  size:1;
signed:0;
         field:int common_pid;                     offset:4;  size:4;
signed:1;
         field:int common_padding;                 offset:8;  size:4;
signed:1;

         field:unsigned long __probe_ip;           offset:12; size:4;
signed:0;
         field:u32 arg1;                           offset:16; size:4;
signed:0;
         field:u32 arg2;                           offset:20; size:4;
signed:0;

    print fmt: "(%lx) arg1=%lx arg2=%lx", REC->__probe_ip, REC->arg1, REC-
>arg2

Right after definition, each event is disabled by default. For tracing
these
events, you need to enable it by:

    # echo 1 > events/uprobes/enable

Lets disable the event after sleeping for some time.

    # sleep 20
    # echo 0 > events/uprobes/enable

And you can see the traced information via /sys/kernel/debug/tracing/trace.

    # cat trace
    # tracer: nop
    #
    #           TASK-PID    CPU#    TIMESTAMP  FUNCTION
    #              | |       |          |         |
                 zsh-24842 [006] 258544.995456: zfree_entry: (0x446420)
arg1=446420 arg2=79
                 zsh-24842 [007] 258545.000270: zfree_exit:  (0x446540 <-
0x446420) arg1=446540 arg2=0
                 zsh-24842 [002] 258545.043929: zfree_entry: (0x446420)
arg1=446420 arg2=79
                 zsh-24842 [004] 258547.046129: zfree_exit:  (0x446540 <-
0x446420) arg1=446540 arg2=0
```

```
156
157 Output shows us uprobe was triggered for a pid 24842 with ip being 0x446420
158 and contents of ax register being 79. And uretprobe was triggered with ip
    at
159 0x446540 with counterpart function entry at 0x446420.
```

Uprobe in a brief example

We explain how Uprobe works by monitoring bash commands as an example. First, we need to identify where the eBPF user hook point it. We can do this by compiling our program with `g++/gcc -g` flag and then run the gdb debugging with it:

```
1 $ gdb /bin/bash
2 (gdb)p readline
3 $1 = {<text variable, no debug info>} 0xd5690 <readline>
```

`readline` is an executable symbol within bash that reads the user command and argument. Based on the gdb result, the readline is located at 0xd5690. I'm not sure how this actually works but the location number does not change if you re-compile the program, therefore it's deterministic.

Next, we add a uprobe to the system monitoring the bash readline function, we need to enter root mode to execute eBPF commands:

```
1 sudo su -
```

```
1 echo 'r:bashReadline /bin/bash:0xd5690 cmd=+0($retval):string' >>
  /sys/kernel/tracing/uprobe_events
2
3 echo 1 > /sys/kernel/tracing/events/uprobes/bashReadline/enable
4
```

The first command creates a uprobe event named `bashReadline`, the uprobe is located at the readline symbol address. The command is return the return value as string. This is written to `/sys/kernel/tracing/uprobe_events`. Next we enable the event by writing `1` to `/sys/kernel/tracing/events/uprobes/bashReadline/enable`.

To monitor the eBPF function, we use the default entry point of the system:

```
1 cat /sys/kernel/tracing/trace_pipe
```

Next, the output of this uprobe monitoring event does not show much useful information, therefore we can write a eBPF program to print out things in a much nicer format. Although one can write a normal C/C++ eBPF program, we use a much simpler way -- eBPF python wrapper. First we need to install the python eBPF wrapper library:

```
1 # pip3 install bcc does not work
2 sudo apt-get install python3-bpfcc # source link:
  https://github.com/iovisor/bcc/issues/2278
```

Next we write a simple eBPF program that prints the command and the parameter:

```
#!/usr/bin/python3

from bcc import BPF
from time import sleep


# load BPF program
bpf_text="""
#include <linux/sched.h>

int printForRoot(struct pt_regs *ctx){

    char command[16] = {};

    //use a bpf helper to get the user id.
    uid_t uid = bpf_get_current_uid_gid() & 0xffffffff;

    //another bpf helper to read a string in userland
    bpf_probe_read_user_str(&command, sizeof(command), (void
*)PT_REGS_RC(ctx));

    if(uid == 0){
        bpf_trace_printk("Command from root: %s",command);
    }
    return 0;
}
"""


b = BPF(text=bpf_text)
b.attach_uretprobe(name="/bin/bash", sym="readline", fn_name="printForRoot")

while(1):
    sleep(1)
```

We use a endless loop to keep the eBPF program attached. The `attach_uretprobe` command specify the user space binary and simply to monitor, and also specify the eBPF program to run. This approach has a downside because the bpf program is provided by text, therefore it's hard to debug the code itself if anything goes run.

As a result here is the output I get after I monitored the event:

```
 bash-1723    [001] DNZff  1598.807344: bashReadline: (0x5624d34b5015 <-
0x5624d3555690) cmd="ls"
            bash-1723    [001] DNZff  1604.504058: bashReadline:
(0x5624d34b5015 <- 0x5624d3555690) cmd="ls -al"
            bash-1723    [001] DNZff  1611.383836: bashReadline:
(0x5624d34b5015 <- 0x5624d3555690) cmd="ls -al *.out"

```

I also played with writing C eBPF program, one example is the eBPF program which tracks open system calls:

```c
#include <uapi/linux/ptrace.h>
#include <linux/sched.h>

BPF_HASH(syscall_count, u32, u64);

int count_syscalls(struct pt_regs *ctx) {
    u32 pid = bpf_get_current_pid_tgid();
    u64 count = 0, *val;

    val = syscall_count.lookup(&pid);
    if (val) {
        count = *val;
    }
    count++;
    syscall_count.update(&pid, &count);

    return 0;
}
```

Project Goals and Objectives

The primary goal was to develop an eBPF-based tool capable of monitoring user programs in real-time. Objectives included gaining a deep understanding of eBPF, developing system-level programming skills, and creating a practical tool for system administrators to monitor processing latencies.

Methodology

We pick httpserver as a user space example, and explain (1) how we implemented the webserver (2) how we hooked the eBPF symbols (3) Our preliminary evaluation results.

We utilized the BCC (BPF Compiler Collection) toolset for developing our eBPF program. The project was implemented in C and Python, focusing on intercepting system calls and monitoring network traffic.

Implementation

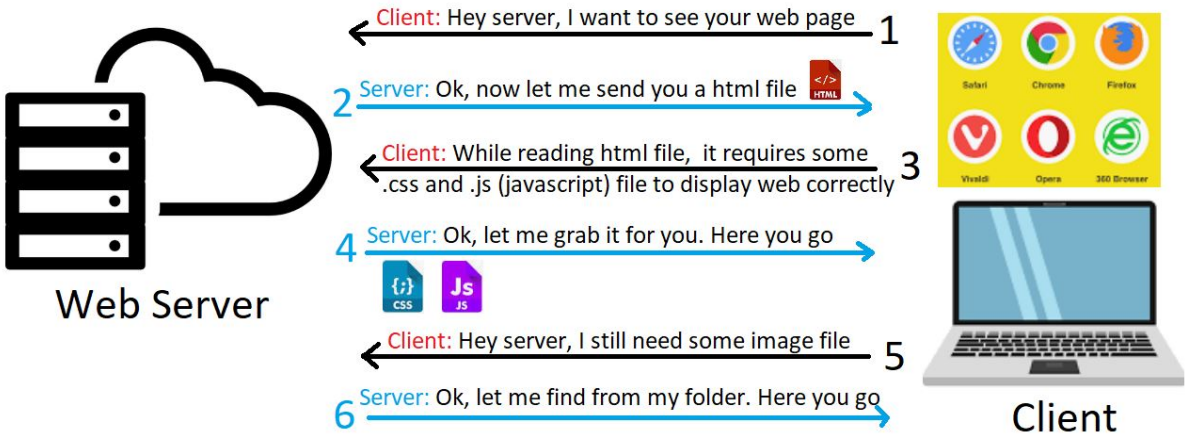
Http Webserver

In the digital realm, servers are powerful systems that provide services to multiple clients. Renowned examples include tech giants like Google, Netflix, and Facebook. These servers interact with clients, which are typically users like us utilizing web browsers such as Chrome, Edge, Opera, or Firefox to communicate with these servers.

Setting up a personal web server is a feasible project. With a basic setup at home, you can create a local network environment, commonly known as a Local Area Network (LAN), using devices like a Wi-Fi router. In this setup, your laptop or a desktop computer can function as the server. This server, ideally running a Linux distribution like Ubuntu or Debian, connects to your home router, establishing the LAN.

To access and interact with your home server, various client devices can be used, including smartphones. When you host a webpage on your home server, it's accessible within your LAN. However, if the webpage includes external resources hosted on the Wide Area Network (WAN) - essentially the broader internet - your server will need internet access to fetch and display these resources correctly.

This simple yet effective setup allows for a hands-on experience with web hosting and network communication, providing a practical understanding of how servers and clients interact within and beyond a local network.



WebServer Implementation and documentation

We first give a brief explanation of the webserver code, and then provide source code with detailed documentation.

This C program demonstrates the implementation of a basic web server using socket programming in C. It includes functions for parsing HTTP requests, determining the requested resource type, and sending appropriate responses, including HTML pages, images, and other static files.

1. **Socket Initialization and Binding:** The `main()` function initializes a TCP socket, binds it to a specified port (8081), and listens for incoming connections. It uses `socket()`, `bind()`, and `listen()` system calls for these purposes. Error handling is implemented for each of these steps.
2. **Connection Handling and Request Processing:** Upon accepting a connection, the server forks a child process to handle the request, allowing the main process to continue listening for new connections. This concurrency model enables the server to handle multiple requests simultaneously.
3. **HTTP Request Parsing:** The `parse()`, `parse_method()`, and `find_token()` functions extract different parts of the HTTP request. `parse()` and `parse_method()` are used to retrieve the requested path and the HTTP method (GET, POST, etc.), respectively, whereas `find_token()` searches for specific tokens within the request header.
4. **Content-Type Determination:** Based on the file extension of the requested resource, the server determines the MIME type of the response (e.g., text/html, image/jpeg). This is crucial for the client (browser) to correctly interpret and display the content.

5. **Resource Serving:** The `send_message()` function is responsible for sending the requested files to the client. It uses the `sendfile()` system call to efficiently send file data over a socket. This function is designed to handle different types of files, including HTML, JPEG images, CSS, JavaScript, and font files.

```c
// Server-side C program to demonstrate basic socket programming
#include <stdio.h>
#include <sys/socket.h>
#include <unistd.h>
#include <stdlib.h>
#include <netinet/in.h>
#include <string.h>
#include <sys/types.h>
#include <fcntl.h>
#include <sys/sendfile.h>
#include <sys/stat.h>
#include <errno.h>

#define PORT 8081

// Function prototypes
char* parse(char line[], const char symbol[]);
char* parse_method(char line[], const char symbol[]);
char* find_token(char line[], const char symbol[], const char match[]);
int send_message(int fd, char image_path[], char head[]);

char http_header[25] = "HTTP/1.1 200 Ok\r\n";

int main(int argc, char const *argv[])
{
    // Server socket descriptor, client socket descriptor, and process ID
    int server_fd, new_socket, pid;
    long valread;
    struct sockaddr_in address; // Server address
    int addrlen = sizeof(address);

    // Creating socket file descriptor
    if ((server_fd = socket(AF_INET, SOCK_STREAM, 0)) == 0)
    {
        perror("In socket creation");
        exit(EXIT_FAILURE);
    }

    // Setting up the server address structure
    address.sin_family = AF_INET;
    address.sin_addr.s_addr = INADDR_ANY;
    address.sin_port = htons(PORT); // Convert port number to network byte order
    memset(address.sin_zero, '\0', sizeof address.sin_zero);

    // Binding the socket to the server address
    if (bind(server_fd, (struct sockaddr *)&address, sizeof(address)) < 0)
    {
        perror("In socket bind");
        close(server_fd);
```

```
        exit(EXIT_FAILURE);
    }

    // Listening for incoming connections
    if (listen(server_fd, 10) < 0)
    {
        perror("In socket listen");
        exit(EXIT_FAILURE);
    }

    // Server main loop
    while(1)
    {
        printf("\n+++++++ Waiting for new connection ++++++++\n\n");
        // Accepting new connection
        if ((new_socket = accept(server_fd, (struct sockaddr *)&address,
(socklen_t*)&addrlen)) < 0)
        {
            perror("In connection accept");
            exit(EXIT_FAILURE);
        }

        // Fork a new process to handle the request
        pid = fork();
        if(pid < 0){
            perror("Error in fork");
            exit(EXIT_FAILURE);
        }

        if(pid == 0){ // Child process
            char buffer[30000] = {0};
            valread = read(new_socket, buffer, 30000); // Read client
request

            // Parsing HTTP request
            char *parse_string_method = parse_method(buffer, " "); // Parse
HTTP method
            char *parse_string = parse(buffer, " "); // Parse requested
path

            // File extension parsing and content type determination
            char *copy = (char *)malloc(strlen(parse_string) + 1);
            strcpy(copy, parse_string);
            char *parse_ext = parse(copy, "."); // Parse file extension

            char *copy_head = (char *)malloc(strlen(http_header) + 200);
            strcpy(copy_head, http_header);

            // Handling GET requests
            if(parse_string_method[0] == 'G' && parse_string_method[1] ==
'E' && parse_string_method[2] == 'T'){
                // Logic to serve different file types based on extension
                // ...

                // Example: Handling JPEG images
```

```
                if ((parse_ext[0] == 'j' && parse_ext[1] == 'p' &&
parse_ext[2] == 'g') ||
                    (parse_ext[0] == 'J' && parse_ext[1] == 'P' &&
parse_ext[2] == 'G')) {
                    // Serve JPEG image
                    // ...
                }

                // Logic for other file types (CSS, JavaScript, etc.)
                // ...

                printf("\n------------------Server sent response----------
----------\n");
            }

            // Handling POST requests
            else if (parse_string_method[0] == 'P' &&
parse_string_method[1] == 'O' && parse_string_method[2] == 'S' &&
parse_string_method[3] == 'T'){
                // Handling POST request logic
                // ...
            }

            close(new_socket); // Close client

```

eBPF hook explained

The process of using eBPF to monitor latency is to label out the start and end point of the process, mark them as function symbols.

```
void start_eBPF_monitor(){
    (void)0;
}

void end_eBPF_monitor(){
    (void)0;
}

void monitored_function() {
    start_eBPF_monitor()
    // Key process here
    end_eBPF_monitor()
}
```

By adding the hooks, we can easily use GDB to identify the address of the function `start_eBPF_monitor` and `end_eBPF_monitor`. The next step will be simply adding the monitor event to eBPF following the same procedure.

Evaluation

We now run the webserver program with the eBPF hook we added:

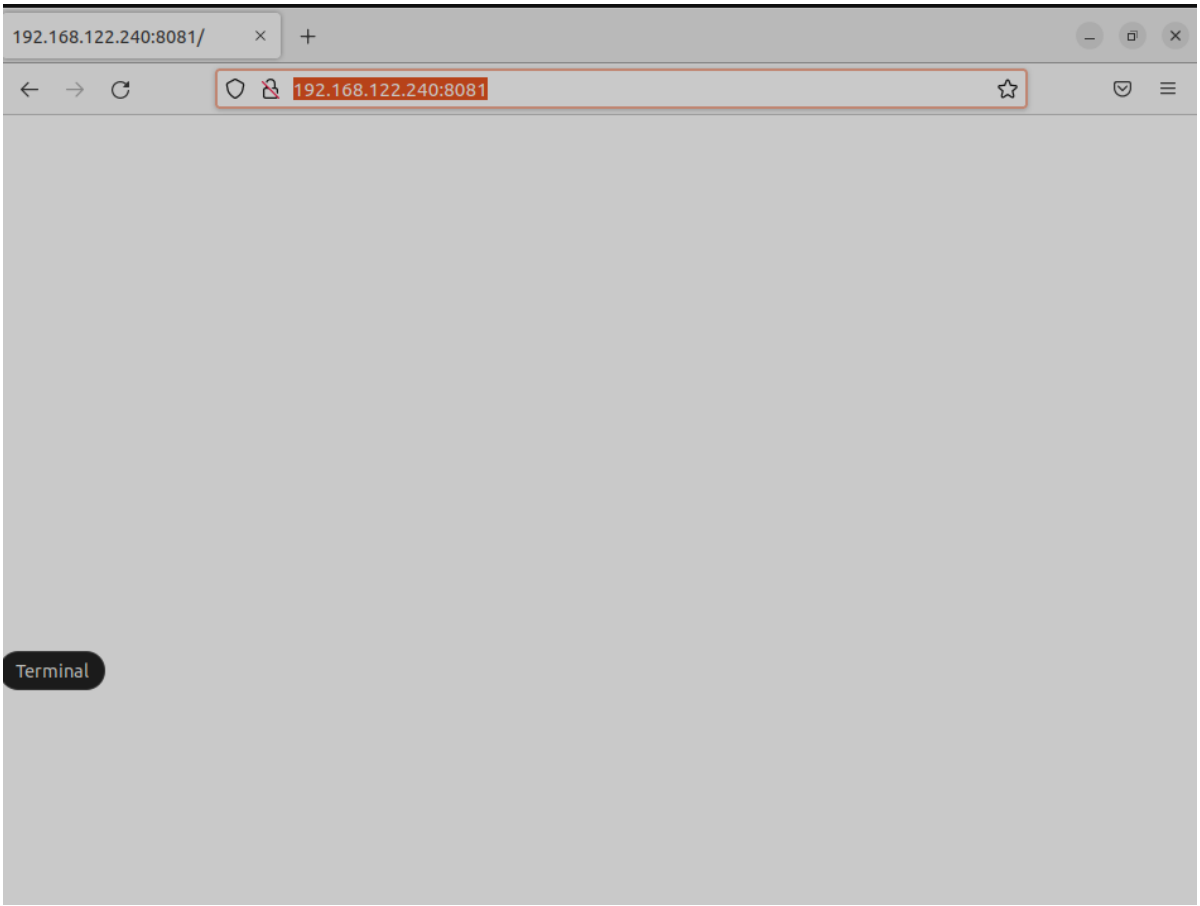
```
>>>>>>>>>>Parent create child with pid: 58277 <<<<<<<<<
+++++++ Waiting for new connection ++++++++


 buffer message: GET / HTTP/1.1
Host: 192.168.122.240:8081
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:105.0) Gecko/20100101 Fir
efox/105.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/w
ebp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
Upgrade-Insecure-Requests: 1


 Client method: GET
Client ask for path: /
Cannot Open file path : ./index.html with error -1

------------------Server sent--------------------------------------------------
-
```

We access the webserver by directly accessing the `IP:port` through firefox web browser:



Next we observe the eBPF output:

We can easily use the difference between two time stamps to estimate the latency of a `send_message` in webserver is around 7ms.