Detecting Malicious Communication in Air-Gap Systems

-

USB monitoring realization

Kelly Zhang

Version

Version	Name	Changes
1.0	Kelly Zhang	Everything : context, realization(what is the goal, important parts video demo), managing &control
1.1	Kelly Zhang	Beperking + additional changes

Inhoud

Context

Air-gap systems, designed to be isolated from external networks, are widely regarded as one of the most secure configurations for protecting sensitive data and critical infrastructure. Physical separation from the internet or other networks significantly reduces the risk of direct cyberattacks. However, this isolation does not guarantee absolute security, as sophisticated adversaries have increasingly found ways to exploit vulnerabilities through indirect communication channels. Malicious communication, often hidden or disguised within normal system operations, poses a growing threat to air-gap systems, making traditional security measures insufficient.

To protect air-gapped systems, we made an analysis document where we compared hardware-based and software-based detection systems. From that, we found out that both systems play an important role in protecting an air-gapped setup. In the document, we also gave several recommendations to build a software-based malware detection system for air-gapped environments — without depending on traditional cloud-based antivirus solutions.

One such recommendation is: The system should detect data being written to a USB from the air-gapped machine, because malware present on an air-gapped system can replicate itself and extract data using USB flash drives to collect sensitive information such as credentials, screenshots, and more. This type of attack is often the only method of exfiltration and infection.

In this document we will go through the realization part of this recommendation. You will first get a small description what the code exactly does and some important part of the code with snippets of the code and a video demo. At last we will we also going through how I managed the code by using github.

Realization

What is the goal ?

This is a Go written program that automatically detects USB-sticks plugged into the system using lsblk. Next each mountpoint of the USB stick will be monitored with help of the linux subsystem fanotify, with this we can detect the following operations : open/read, write, create, delete, moved from, moved to, rename, attribute change, write close and read close on files in the USB-stick.

Picture below is an example of the output of the program. We can see it detected a usb with the mountpoint /media/test/USB DISK and detected open operation and the read clos operation, at the end we plugged the usb out and it stopped monitoring.

```
Starting to monitor for USBs:
/media/test/USB DISK is starting
Monitoring /media/test/USB DISK...
[09:43:54][/media/test/USB DISK]Open detected from PID: 323064
[09:43:54][/media/test/USB DISK]Read close detected from PID: 323064
Stopped monitoring on /media/test/USB DISK
```

Important parts of the program :

1. **C-integration for Fanotify**

```
/*
#include <stdlib.h>
#include <sys/fanotify.h>
#include <fcntl.h>
#include <unistd.h>
#include <errno.h>

static int getErrno() {
    return errno;
}
*/
import "C"
```

At the beginning of the program we have a C-block that gets C function for example fanotify_init, fanotify_mark, en errno. These are very important because fanotify only accepts C- system calls. That's why we have some C in our code, examples :

```
ret, err := C.fanotify_mark(
        m.fd,
        C.FAN_MARK_ADD,
        C.FAN_CREATE|C.FAN_OPEN|C.FAN_MODIFY|
                C.FAN_MOVED_TO|C.FAN_MOVED_FROM|C.FAN_RENAME|
                C.FAN_DELETE|C.FAN_ATTRIB|C.FAN_CLOSE_WRITE|
                C.FAN_CLOSE_NOWRITE|C.FAN_ONDIR|C.FAN_EVENT_ON_CHILD,
        C.AT_FDCWD,
        cPath.
```

2. Detectie van USB-sticks

In the function DetectNewUSB() we are using lsblk- command to get the blockdevices on the system. `cmd := exec.Command("lsblk", "-J", "-o", "NAME,MOUNTPOINTS,TRAN")` However we only want `if device.Tran != nil && *device.Tran == "usb"` usb devices, so we only take data out when "tran": "usb" . We take all the partitions and mountpoints out and put it inside a struct called USB.

3. Verschil-analyse van USB's

The USBDifferenceChecker() function compares whether new USB drives have been connected compared to the previous detection cycle, using a unique key generated by the Key() function.

```
for _, usb := range u.OutputUSB {
    if !existingMap[usb.Key()] {
        newUSBs = append(newUSBs, usb)
    }
}
```

The Key() function returns a key that consists of the device name, the number of partitions, and a hash of all mount points.
This means that if new mount points or partitions are added by malware, the change can still be detected, and the corresponding mount point can be monitored.

```
return fmt.Sprintf("%s-%d-%s", u.Name, len(u.Partitions), hash)
```

4. Monitoring each mountpoint

For each new USB partition, a separate Monitor struct is created using NewMonitor(). In this function, Fanotify is initialized and check if it is a valid path and directory, at last the file descriptor is added to the Monitor struct.

```
        fd, err := C.fanotify_init(C.FAN_REPORT_FID|C.FAN_REPORT_DFID_NAME|C.FAN_CLASS_NOTIF|C.FAN_NONBLOCK,
C.O_RDONLY)
```

The Start() function marks the mount point using fanotify_mark() and then listens for incoming events via read() on the fanotify file descriptor.
Events such as FAN_OPEN, FAN_CREATE, FAN_DELETE, etc., are logged along with a timestamp and the PID of the triggering process (an example of the output is shown under *"What is the goal?"*).

```
ret, err := C.fanotify_mark(
        m.fd,
        C.FAN_MARK_ADD,
        C.FAN_CREATE|C.FAN_OPEN|C.FAN_MODIFY|
                C.FAN_MOVED_TO|C.FAN_MOVED_FROM|C.FAN_RENAME|
                C.FAN_DELETE|C.FAN_ATTRIB|C.FAN_CLOSE_WRITE|
                C.FAN_CLOSE_NOWRITE|C.FAN_ONDIR|C.FAN_EVENT_ON_CHILD,
        C.AT_FDCWD,
        cPath,
)
```

5. Mountpoint-check

To verify whether the mount point is still active (e.g., when the USB is removed), the MountpointChecker() function is executed, which returns either true or false. MountpointChecker() performs a new lsblk scan and checks whether the current mount point is still present in the list returned by the new lsblk scan.

```
if mountpoint == m.Mountpath {
        //fmt.Println("found")
        return true, nil
}
```

Beperkingen

Unfortunately, using Fanotify requires root privileges, which is understandable given that it can detect many file operations and, when using FAN_CLASS_PRE_CONTENT, even block them.

Additionally, there are other methods available to block actions besides Fanotify, but due to time constraints, these could not be explored.

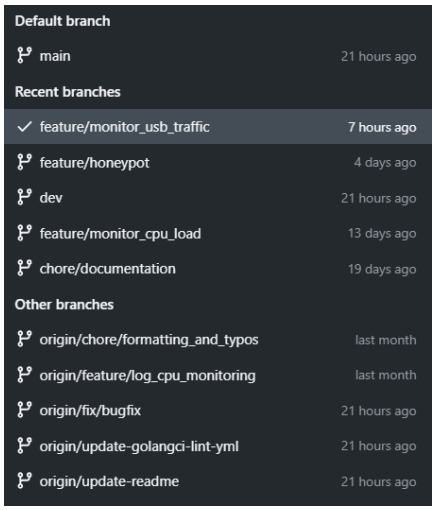
Additional changes:
Since my teammate Nahit was working on unit testing, some changes were made to the NewUSBMonitor function.
Instead of initializing directly, a separate function named Initialize was created along with a corresponding struct to support dependency injection. The main functionality is still the same.

Video demo :

https://youtube.com/shorts/YpQDjZcLFPc?si=IaT_lFtAv-F9A2cq

Manage &control

Github link : https://github.com/Dogru-Isim/airgapantivirus/blob/dev/internal/monitoring/usb_monitoring_linux.go



Tijdens het realiseren hadden we een github project gemaakt met verschillende brancehs. Ik werkte het meeste aan de branch feature/monitor_usb_traffic waar ik aan deze progamma heb gewerkt.