Container-Aware Exploit Detection and False Positive Reduction

Julia Odden
Northwestern University

Braden Svoboda
Northwestern University

Yuanjun Zhang
Northwestern University

Abstract

Containerization and container systems, such as Docker and Kubernetes, are becoming increasingly popular as an alternative to full virtualization because they are lightweight, flexible, and efficient. Containers use several isolation mechanisms, namely namespace isolation and cgroups, to separate the processes and memory belonging to the container from those of the host machine. Namespace isolation leads to potentially abnormal behavior when attacks that normally target entire host machines begin within a container environment.

1 Introduction

Ming Yan, an ongoing intrusion detection system (IDS) project at Zhejiang University, China, is a provenance-based advanced persistent threat (APT) detection software that was not designed to detect or prevent attacks made against a host machine from within a containerized environment. We and the project developers believe that it is important for any robust IDS to have mechanisms for identifying malicious behaviors in containerized environments. We make three primary contributions to the project:

1. An analysis of the performance of the Ming Yan IDS on attacks made from within a containerized environment.
2. Modifications to the Ming Yan client source code to make the system more robust to container-based attacks.
3. Policies to handle the resulting false positives generated when we make the IDS more sensitive.

Our modifications are running on a single client instance of the Ming Yan IDS, and we also present initial benchmarking of that modified client. Our modifications are detailed in Appendix 1.

2 Background and Motivation

Many modern intrusion detection systems are not built with awareness of tools used to create container environments, like Linux namespaces and union filesystems, among others. This potential lack of awareness creates uncertainty about whether these IDSs can detect risky and malicious behavior in container namespaces. While most container engines provide their own security mechanisms, and many leverage the security mechanisms of their host operating system (such as Docker using seccomp, a Linux security facility), these mechanisms are not enough to guarantee security. Users need an IDS that can effectively audit the security of these containers in real time.

The principal isolation mechanisms used by the Docker container engine on the Linux platform are namespaces and cgroups. Namespaces allow processes to exist and be managed by the kernel under different groups, effectively allowing the kernel to “sandbox” the processes by limiting how they interact with other namespaces. The five namespaces used by Docker are process ID (PID), file system (MNT), interprocess communication space (IPC), Unix time sharing (UTC), and network (NET). Cgroups are closely related to namespaces and allow the kernel to limit the amount of resources collectively given to a process or group of processes. For instance, a user could limit the CPU usage of a container to .5, or 50% of the overall CPU. It is also possible to limit the amount of memory used or IO throughput to devices. The more isolated and limited a container is, the more secure it is, but this comes at the cost of making it difficult for IDSs running in the host namespace to effectively track behavior inside of the container.

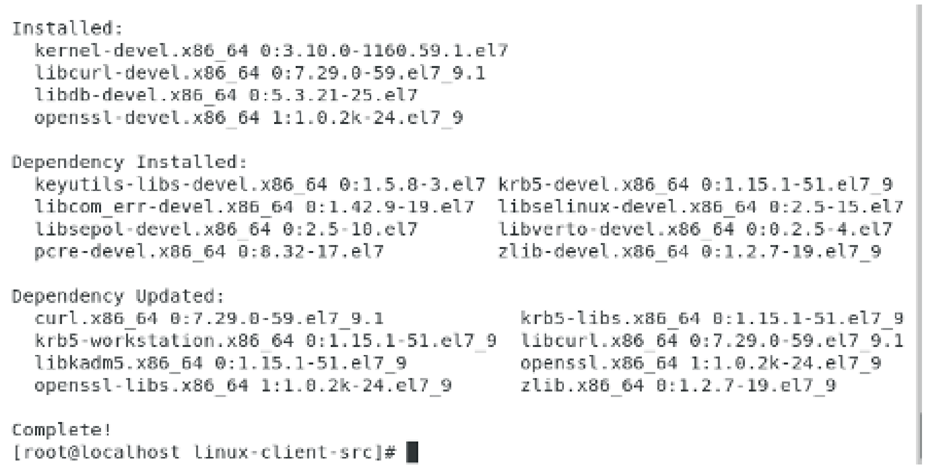
3 IDS Software Installation

Ming Yan uses a client-server architecture, where the client sits on the user machine and periodically sends logs to the server. The client comes in two flavors: first, the “release” client installed via a curl command to the server; and second, a source-code installation. Prior to our modifications, the source code installation and release installation of the client were functionally identical. The first challenge we addressed was the installation of the Ming Yan client and server. All of the instances of the clients and the server were installed on the same host within VirtualBox virtual machines.

3.1 Client

Our client ran on a 7.9.2009 CentOS virtual machine (VM) with 2GB of RAM, two Intel Xeon X5560 2.80GHz CPUs, and 32GB of disk space. To download the release client, we simply had to copy and paste a curl command from the server’s web interface, substituting in the server’s IP address. This downloads the client installation executable directly from the server.

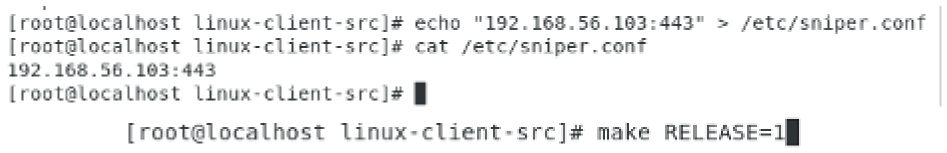
To install the client from source, we moved the source code directory onto the virtual machine and installed several Linux kernel packages. The source-code client ran on a different virtual machine with the same specifications as the release client’s virtual machine.



```
Installed:
  kernel-devel.x86_64 0:3.10.0-1160.59.1.el7
  libcurl-devel.x86_64 0:7.29.0-59.el7_9.1
  libdb-devel.x86_64 0:5.3.21-25.el7
  openssl-devel.x86_64 1:1.0.2k-24.el7_9

Dependency Installed:
  keyutils-libs-devel.x86_64 0:1.5.8-3.el7 krb5-devel.x86_64 0:1.15.1-51.el7_9
  libcom_err-devel.x86_64 0:1.42.9-19.el7  libselinux-devel.x86_64 0:2.5-15.el7
  libsepol-devel.x86_64 0:2.5-10.el7       libverto-devel.x86_64 0:0.2.5-4.el7
  pcre-devel.x86_64 0:8.32-17.el7          zlib-devel.x86_64 0:1.2.7-19.el7_9

Dependency Updated:
  curl.x86_64 0:7.29.0-59.el7_9.1             krb5-libs.x86_64 0:1.15.1-51.el7_9
  krb5-workstation.x86_64 0:1.15.1-51.el7_9   libcurl.x86_64 0:7.29.0-59.el7_9.1
  libkadm5.x86_64 0:1.15.1-51.el7_9           openssl.x86_64 1:1.0.2k-24.el7_9
  openssl-libs.x86_64 1:1.0.2k-24.el7_9       zlib.x86_64 0:1.2.7-19.el7_9

Complete!
[root@localhost linux-client-src]# █
```

Figure 1: Installation of the required Linux kernel packages.

The source code client uses an executable called "sniper" to orchestrate most of its functionality. We set up sniper's configuration by replacing the control IP address with our actual Server IP and changed the compilation flags from "all: user kernel tray" to "all:user kernel" in the Makefile. This change is detailed in Appendix 1.

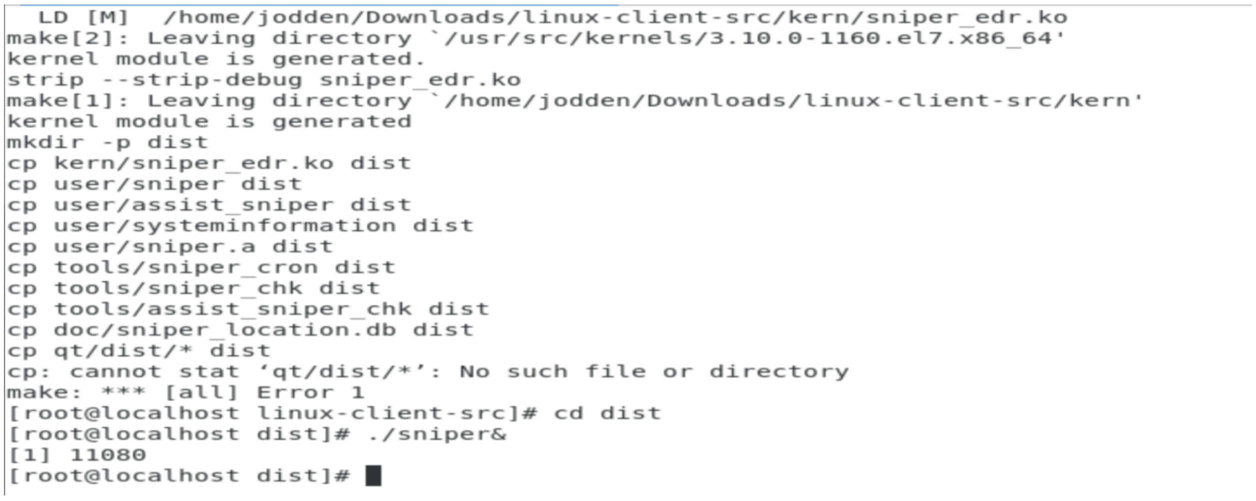


```
[root@localhost linux-client-src]# echo "192.168.56.103:443" > /etc/sniper.conf
[root@localhost linux-client-src]# cat /etc/sniper.conf
192.168.56.103:443
[root@localhost linux-client-src]# 
```

```
[root@localhost linux-client-src]# make RELEASE=1
```

Figure 2: Configuration and compilation of the sniper executable.

When we attempted to compile the code we were given, we received an error stating that it was missing a required directory, qt/dist, containing package dependencies. This error can be ignored.



```
  LD [M]  /home/jodden/Downloads/linux-client-src/kern/sniper_edr.ko
make[2]: Leaving directory `/usr/src/kernels/3.10.0-1160.el7.x86_64'
kernel module is generated.
strip --strip-debug sniper_edr.ko
make[1]: Leaving directory `/home/jodden/Downloads/linux-client-src/kern'
kernel module is generated
mkdir -p dist
cp kern/sniper_edr.ko dist
cp user/sniper dist
cp user/assist_sniper dist
cp user/systeminformation dist
cp user/sniper.a dist
cp tools/sniper_cron dist
cp tools/sniper_chk dist
cp tools/assist_sniper_chk dist
cp doc/sniper_location.db dist
cp qt/dist/* dist
cp: cannot stat 'qt/dist/*': No such file or directory
make: *** [all] Error 1
[root@localhost linux-client-src]# cd dist
[root@localhost dist]# ./sniper&
[1] 11080
[root@localhost dist]# 
```

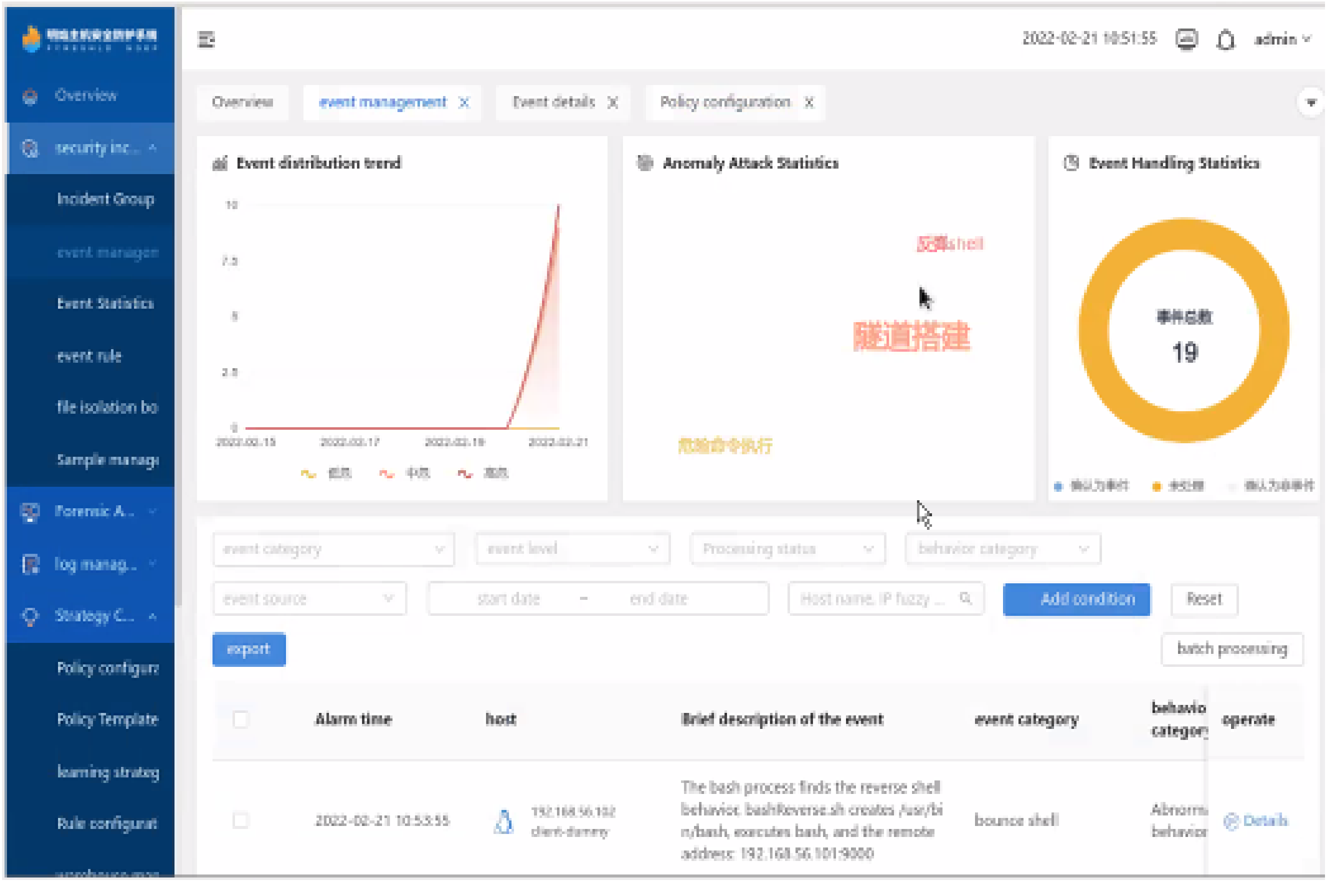
Figure 3: Output from `make` and subsequent execution of sniper.

We note that our installation package was simplified by the project developers to reduce configuration time. After we ignore this error and open the `dist` directory under the current environment, we see that the sniper executable did indeed compile successfully. We can now execute the sniper process and observe that the client has connected to the server through messages printed to console during normal operations.

3.2 Server

Our server instance is hosted on a virtual machine running a CentOS 7.8.2003 installation. This VM was given three Intel Xeon X5560 2.80GHz CPUs, 20 GB of RAM, and 100 GB of disk space. While these values are all significantly lower than the recommended value for using Ming Yan in production, since we were using the server for simpler testing procedures and managing only four clients, we deemed this acceptable for our use. All the communication between the server and the clients went through a virtual ethernet (veth) interface based on the host machine.

We configured the virtual machines to use static IPs and ensured that the server VM was only active on the host-only veth interface, a second installation fixed the previous issues. Below is an example of the server's control interface, a locally hosted webpage, following successful installation and configuration of a client.



	Alarm time	host	Brief description of the event	event category	behavio categor	operate
	2022-02-21 10:53:55	192.168.56.102 client-dummy	The bash process finds the reverse shell behavior. bashReverse.sh creates /usr/bin/bash, executes bash, and the remote address: 192.168.56.101:9000	bounce shell	Abnorm behavior	Details

Figure 4: First glimpse of the server's web interface.

As a final note about the server, we were asked to set a security policy to be used with our clients. Following advice from our mentors, we set the most restrictive policy possible, enabling all detection methods in hopes that anything malicious would be caught and reported by the clients. Our policy was set to be used on all the clients. This later became important in investigating false positives as well. The updates we made to the security policy of our modified client are detailed in Appendix 1.

4 Benchmarking

Our first goal was to determine how well Ming Yan's release client performed with regards to identifying exploits initiated within a container environment. We wanted to measure two characteristics: first, the rate at which Ming Yan could successfully identify malicious behavior; and second, the rate at which it raised false alarms for normal workloads.

4.1 Test Exploits

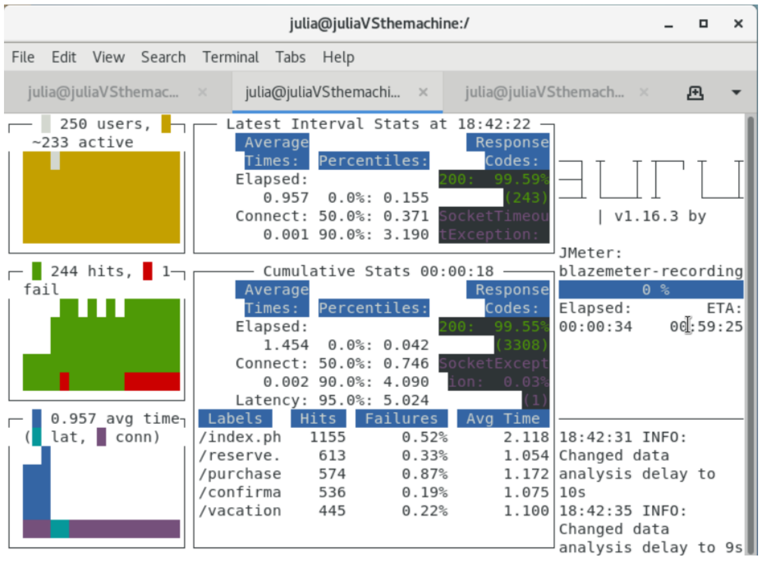
The project developers provided us with a set of test scripts that would perform a variety of exploits ranging from reverse shells to simple deletions of privileged files. Table 1 summarizes our findings. The one exploit that we flagged as having been detected on the host machine but not within Docker is the linux-abnormal program execution-on-alarm. Specifically, the behavior we executed was to create a non-malicious executable file within the /tmp directory of the host machine (ours simply echoed "Hello, world!" to the command line). We then initialized a Docker container that shared the /tmp directory with the host machine and ran the executable from within the container. This behavior, while not technically malicious, is suspicious, and our alarm policy on Ming Yan was configured such that this behavior should have been detected and flagged. Ming Yan successfully detected each of the other attacks from both on the bare-metal host and from within the Docker container. More details about each of these tests can be found here.

Behavior	Detected on the host machine	Detected within Docker
linux-malicious mining-alarm	Yes	Yes
linux-abnormal program execution-on-alarm	Yes	No
linux-MBR protection (dd)-alarm	Yes	Yes
linux-tunnel building-alarm-socat-TCP	Yes	Yes, but only with shared network namespace
linux-Illegal Internet Connection-Alarm	Yes	Yes
linux-bounce shell-bash bounce	Yes	Yes, but only with shared network namespace

Table 1: We ran each suggested exploit on both the host machine and from within a Docker container and documented whether Ming Yan detected it in each scenario. Note that when the linux-tunnel building-alarm-socat-TCP and linux-bounce shell-bash bounce exploits were run without shared network namespace (i.e., when the exploits were run without the –net=host flag), those exploits were not detected from within Docker.

4.2 Standard Workload

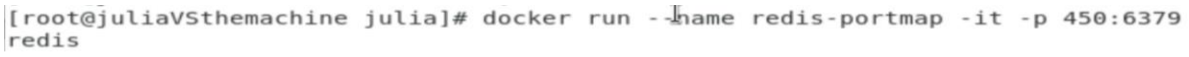
To verify that Ming Yan would not generate false positives on a standard workload from within a Docker container, we installed the blazemeter/taurus Docker image and ran several tests of varying lengths and intensities from within the containers. The server and the release client did not detect any false positives generated by these workloads. We concluded from this that Ming Yan correctly permits most standard network and I/O workloads from within containers.



```
  250 users,          Latest Interval Stats at 18:42:22
~233 active             Average                Response
                         Times:  Percentiles:    Codes:
                       Elapsed:                200:  99.59%
                          0.957  0.0%: 0.155          (243)
                       Connect: 50.0%: 0.371  SocketTimeou     | v1.16.3 by
                          0.001 90.0%: 3.190  tException:
                                                              JMeter:
  244 hits,  1          Cumulative Stats 00:00:18             blazemeter-recording
fail                    Average                Response              0 %
                         Times:  Percentiles:    Codes:      Elapsed:        ETA:
                       Elapsed:                200:  99.55%  00:00:34    00:59:25
                          1.454  0.0%: 0.042         (3308)
                       Connect: 50.0%: 0.746  SocketExcept
                          0.002 90.0%: 4.090   ion:  0.03%
                       Latency: 95.0%: 5.024            (1)
  0.957 avg time      Labels     Hits   Failures   Avg Time
( lat,  conn)         /index.ph  1155      0.52%      2.118  18:42:31 INFO:
                      /reserve.   613      0.33%      1.054  Changed data
                      /purchase   574      0.87%      1.172  analysis delay to
                      /confirma   536      0.19%      1.075  10s
                      /vacation   445      0.22%      1.100  18:42:35 INFO:
                                                             Changed data
                                                             analysis delay to 9s
```

Figure 5: A snapshot of the sample workload using blazemeter/taurus.

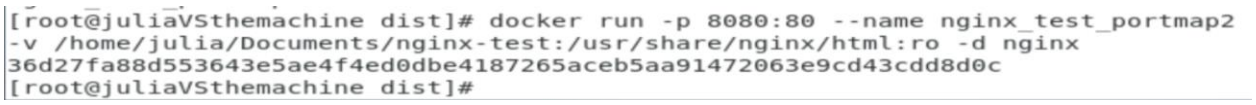
We also initialized a redis container and used it lightly, which also did not create false positives, even when we remotely connected to the container and killed the redis instance without authorization. Ming Yan correctly did not report any of these connections as malicious, including the shutdown signal; although the command was questionable, it was executed through legal channels, so Ming Yan had no reason to flag it as malicious.



```
[root@juliaVSthemachine julia]# docker run --name redis-portmap -it -p 450:6379
redis
```

Figure 6: The redis container runtime configuration.

Finally, we initialized an Nginx container and used it to host a very simple web page. Ming Yan permitted this behavior without issue, although it did (rightly) raise some concerns about the port mapping we performed.

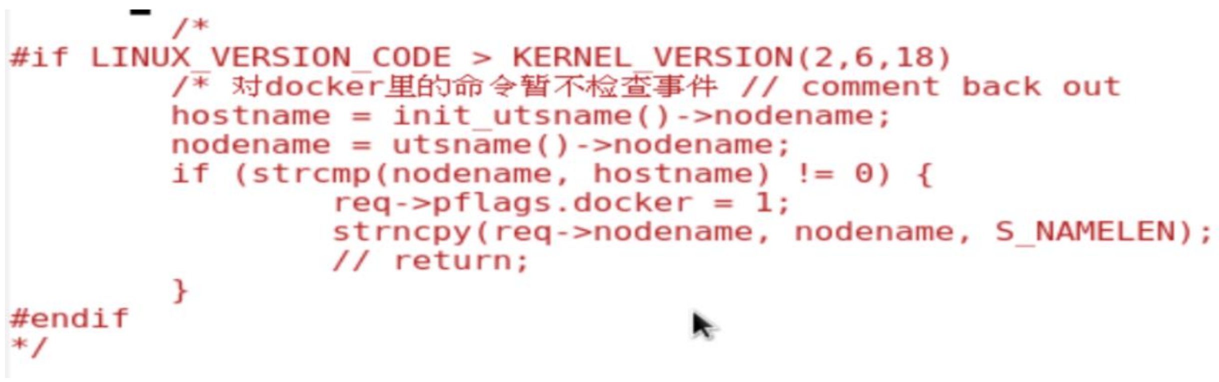


```
[root@juliaVSthemachine dist]# docker run -p 8080:80 --name nginx_test_portmap2
-v /home/julia/Documents/nginx-test:/usr/share/nginx/html:ro -d nginx
36d27fa88d553643e5ae4f4ed0dbe4187265aceb5aa91472063e9cd43cdd8d0c
[root@juliaVSthemachine dist]#
```

Figure 7: The Nginx container runtime configuration. Note the -p 8080:80 flag; this mapped port 8080 to port 80. The web page was accessible internally at https://127.0.0.1:8080 or externally at https://<machine IP>:8080.

5 Code Base Modifications

Because we did not register any false positives, the initial modifications to the IDS were intended to address the single exploit that Ming Yan had failed to detect within the containerized environment: linux-abnormal program execution-on-alarm. The project developers pointed us toward a section in the source code to debug. The snippet below shows the function in the source code of the client that compares the host name to the name of the node and, if they do not match, marks the container flag as 1. Later, the code checks if the container flag is 1, and if it is, it simply ignores the behavior. The reasoning behind this was that Ming Yan was not supposed to detect attacks made within the containers, and the continuous false positives generated from behaviors within the containers created more analysis fatigue than they were worth. The developers made the informed decision to prevent the server from reporting such attacks.



```
        /*
#if LINUX_VERSION_CODE > KERNEL_VERSION(2,6,18)
        /* 对docker里的命令暂不检查事件 // comment back out
        hostname = init_utsname()->nodename;
        nodename = utsname()->nodename;
        if (strcmp(nodename, hostname) != 0) {
                req->pflags.docker = 1;
                strncpy(req->nodename, nodename, S_NAMELEN);
                // return;
        }
#endif
*/
```

Figure 8: The portion of source code that we modified.

As part of our modifications, we removed the block of code that blocked those detections. We then ran the same attack scripts that we ran previously to determine if removing this code would allow the server to detect the linux-abnormal program execution-on-alarm exploit.

Behavior	Detected within Docker by release client	Detected within Docker by modified client
linux-malicious mining-alarm	Yes	Yes
linux-abnormal program execution-on-alarm	No	Yes
linux-MBR protection (dd)-alarm	Yes	Yes
linux-tunnel building-alarm-socat-TCP	Yes, with shared network namespace	Yes
linux-Illegal Internet Connection-Alarm	Yes	Yes
linux-bounce shell-bash bounce	Yes, with shared network namespace	Yes

Table 2: Exploit results using the modified client.

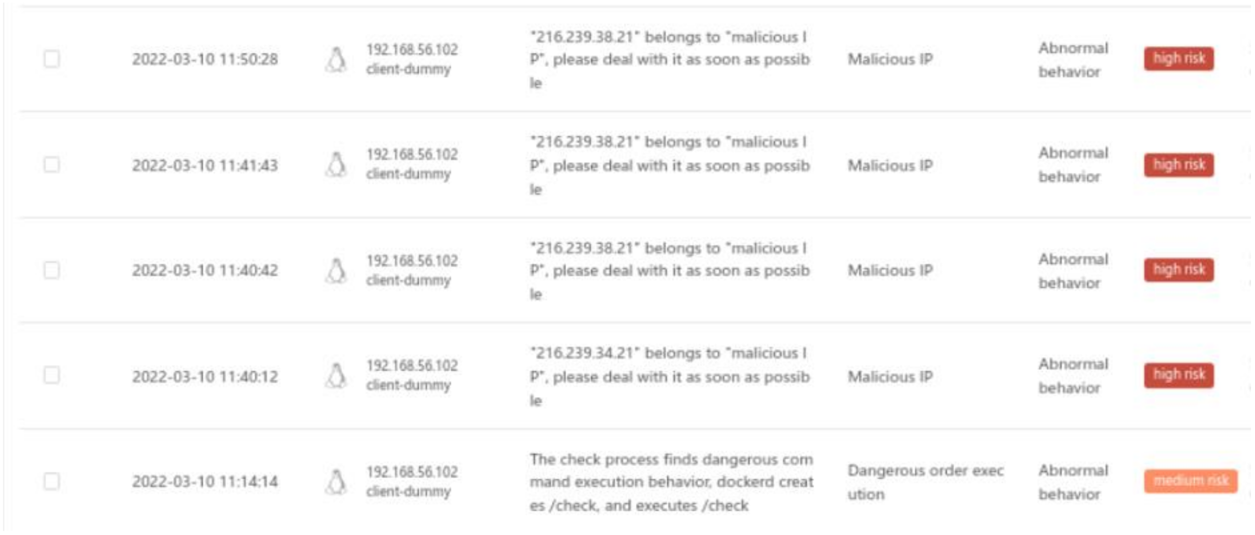
The update to the source code did allow Ming Yan to detect the abnormal execution from within the /tmp directory, without compromising any of the previous correct detections.

6 False Positives

Because we were warned that the code inclusion from section 5 would create false positives, we repeated our benchmarking procedure on the modified client. We repeated the standard workload experiment from section 4.2 on the machine with the modified client and again did not observe any false positives, so we will not discuss it further. Our `redis` workload also did not create any new false positives. While we were not able to fully recreate workloads that real Ming Yan customers might be running, we attempted to, and did notice a few false positives.

6.1 Malicious IP

The Malicious IP false positive was generated by pinging the GitHub homepage of one of our teammates. This false positive is unlike the docker-proxy one generated by using Nginx (described in the following section) because it was replicable on both versions of the client that we had and occurred outside of a container environment.



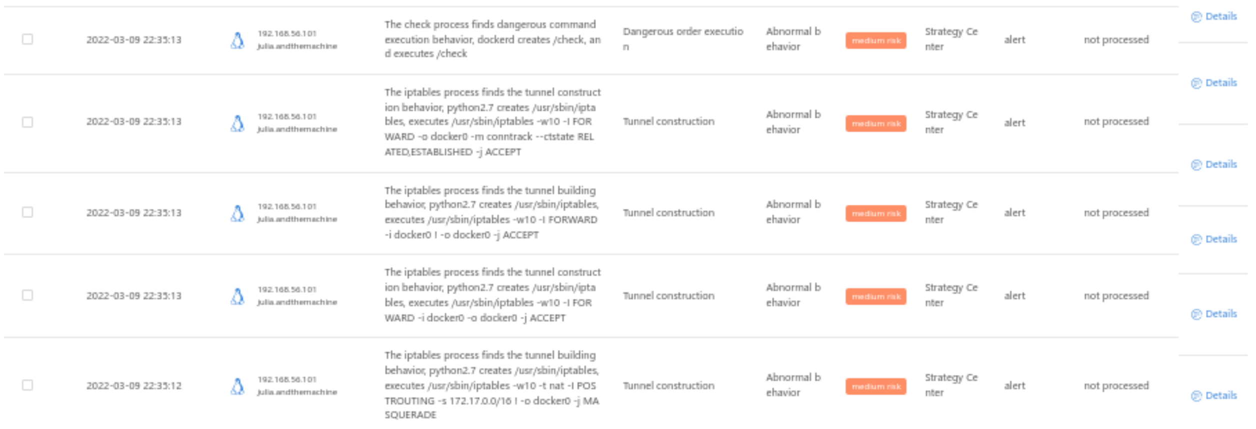
☐	2022-03-10 11:50:28	192.168.56.102 client-dummy	"216.239.38.21" belongs to "malicious IP", please deal with it as soon as possible	Malicious IP	Abnormal behavior	high risk
☐	2022-03-10 11:41:43	192.168.56.102 client-dummy	"216.239.38.21" belongs to "malicious IP", please deal with it as soon as possible	Malicious IP	Abnormal behavior	high risk
☐	2022-03-10 11:40:42	192.168.56.102 client-dummy	"216.239.38.21" belongs to "malicious IP", please deal with it as soon as possible	Malicious IP	Abnormal behavior	high risk
☐	2022-03-10 11:40:12	192.168.56.102 client-dummy	"216.239.34.21" belongs to "malicious IP", please deal with it as soon as possible	Malicious IP	Abnormal behavior	high risk
☐	2022-03-10 11:14:14	192.168.56.102 client-dummy	The check process finds dangerous command execution behavior, dockerd creates /check, and executes /check	Dangerous order execution	Abnormal behavior	medium risk

Figure 9: Server-reported events citing malicious IP addresses.

After consulting the developers about this, we found that the malicious IP detection is managed by an SQLite database storing IP addresses that had previously been classified as malicious by some sort of external audit. This database is then consulted when a user connects to another machine via the internet (or in the LAN) to check whether it is malicious. We are classifying this as a false positive because the IP address in the above screenshot corresponds to the GitHub homepage of a teammate, which is clearly not malicious. It is unclear why this IP address was ever classified as malicious, but without access to the malicious IP database, we were unable to correct this issue. A future area for research might be the IP classification system.

6.2 Docker Startup

Every time we started the Docker container, Ming Yan reported a rash of "Tunnel Construction" abnormal behaviors (fig. 10, below). We classify these alerts as false positives, although it makes sense that they appear; Docker performs several functionalities on startup that do appear suspiciously like the construction of a tunnel. Ming Yan is configured to alert users with every modification to the IP tables on the host machine. Docker does have permission to do this, and indeed, needs to do it in order to function, but most instances in which the IP tables are modified by a process other than the root user are malicious. We did not want to disable the alerts for the IP table modification, but simply whitelisting the Docker startup process (or the Docker application itself) would remove too much sensitivity from the IDS. We did not address this issue, and it could be an area for future research.



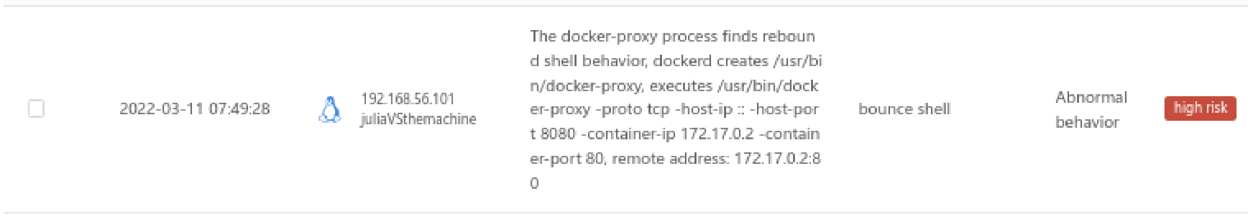
☐	2022-03-09 22:35:13	192.168.56.101 julia.andthemachine	The check process finds dangerous command execution behavior, dockerd creates /check, and executes /check	Dangerous order execution	Abnormal behavior	medium risk	Strategy Center	alert	not processed	Details
☐	2022-03-09 22:35:13	192.168.56.101 julia.andthemachine	The iptables process finds the tunnel construction behavior, python2.7 creates /usr/sbin/iptables, executes /usr/sbin/iptables -w10 -I FORWARD -o docker0 -m conntrack --ctstate RELATED,ESTABLISHED -j ACCEPT	Tunnel construction	Abnormal behavior	medium risk	Strategy Center	alert	not processed	Details
☐	2022-03-09 22:35:13	192.168.56.101 julia.andthemachine	The iptables process finds the tunnel building behavior, python2.7 creates /usr/sbin/iptables, executes /usr/sbin/iptables -w10 -I FORWARD -i docker0 ! -o docker0 -j ACCEPT	Tunnel construction	Abnormal behavior	medium risk	Strategy Center	alert	not processed	Details
☐	2022-03-09 22:35:13	192.168.56.101 julia.andthemachine	The iptables process finds the tunnel construction behavior, python2.7 creates /usr/sbin/iptables, executes /usr/sbin/iptables -w10 -I FORWARD -i docker0 -o docker0 -j ACCEPT	Tunnel construction	Abnormal behavior	medium risk	Strategy Center	alert	not processed	Details
☐	2022-03-09 22:35:12	192.168.56.101 julia.andthemachine	The iptables process finds the tunnel building behavior, python2.7 creates /usr/sbin/iptables, executes /usr/sbin/iptables -w10 -t nat -I POSTROUTING -s 172.17.0.0/16 ! -o docker0 -j MASQUERADE	Tunnel construction	Abnormal behavior	medium risk	Strategy Center	alert	not processed	Details

Figure 10: The false positives from one startup of the Docker process

6.3 Nginx/docker-proxy

One of the services that was used as an example by the developers for something that generated false positives was Nginx. Nginx is an open-source, flexible web server software that we were using to serve a basic HTML page. The actions taken below are on the modified client.

We coded a basic HTML page, pulled the official Nginx Docker image, and initialized a container using that image. The exact command executed is shown in Figure 7. Interestingly, the event did not occur until we attempted to access the HTML page from the same host that was running the container.



☐	2022-03-11 07:49:28	192.168.56.101 juliaVSthemachine	The docker-proxy process finds rebound shell behavior, dockerd creates /usr/bin/docker-proxy, executes /usr/bin/docker-proxy -proto tcp -host-ip :: -host-port 8080 -container-ip 172.17.0.2 -container-port 80, remote address: 172.17.0.2:80	bounce shell	Abnormal behavior	high risk

Figure 11: Event reported by server after running the Nginx container shown in Figure 7 and attempting to access the website at http://localhost:8080.

What this event report indicates is that the process **docker-proxy** is creating malicious reverse shell behavior. Docker-proxy is a process that runs in user space as a child process of dockerd, the Docker daemon, and maps a port on the host machine to a port within the container, forwarding incoming traffic on the host to that port inside the container. With the modified client, this is reported as a high-risk event. However, this is a standard Docker subprocess used for port forwarding and is being used in a non-malicious manner (especially as it is being accessed from the same machine), and therefore we determine that this is a false positive.

Ming Yan is clearly aware of the correct behavior within the container. Fig. 12, below, illustrates the process tree of the so-called malicious action provided by the server. It identifies that the dockerd process was spawned by systemd and subsequently created docker-proxy, which was the flagged process. Fig. 13 shows the output of docker pstree, which matches Ming Yan's output (not shown is the top of the tree, which shows that dockerd is a child process of systemd).

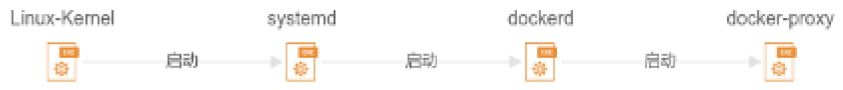
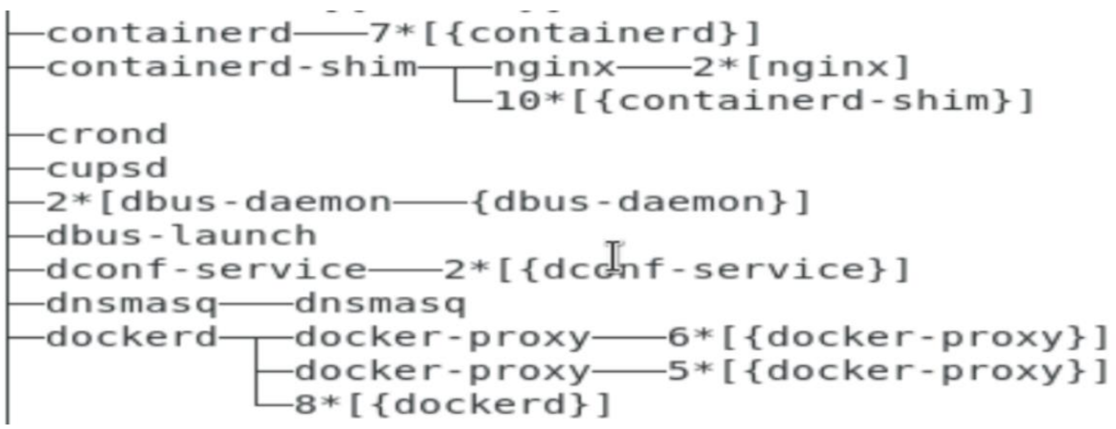


Figure 12: Ming Yan's process tree for docker-proxy



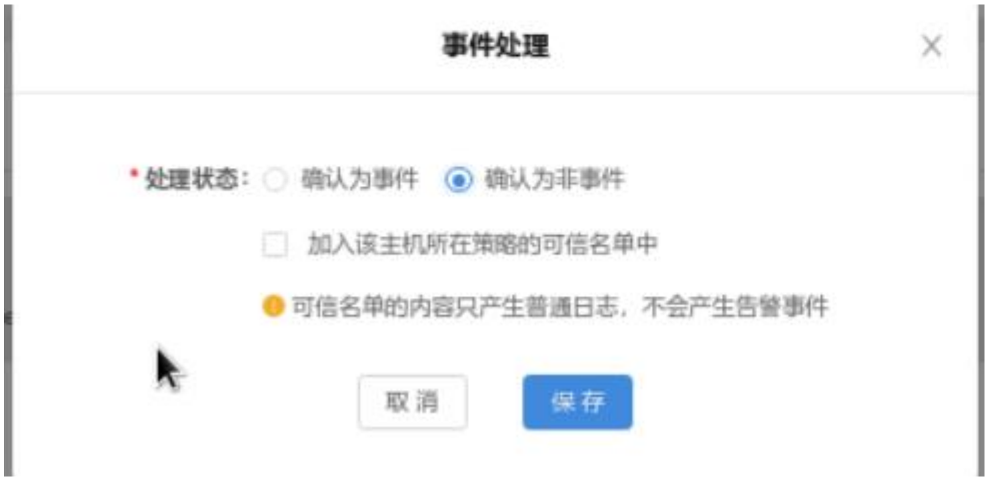
```
├─containerd───7*[{containerd}]
├─containerd-shim─┬─nginx───2*[nginx]
│                 └─10*[{containerd-shim}]
├─crond
├─cupsd
├─2*[dbus-daemon───{dbus-daemon}]
├─dbus-launch
├─dconf-service───2*[{dconf-service}]
├─dnsmasq───dnsmasq
├─dockerd─┬─docker-proxy───6*[{docker-proxy}]
│         ├─docker-proxy───5*[{docker-proxy}]
│         └─8*[{dockerd}]
```

Figure 13: The pstree output on Docker for docker-proxy

7 Policy Modifications to Reduce False Positives

While the original goal was to reduce the false positives by making changes to the client source code, enabling more robust detection, we found that the false positives we detected can be mitigated through additional configurations to the server's policy interface.

To mitigate the docker-proxy false positive, we made the simple change of adding the docker-proxy process to the server's whitelist, meaning that it will not be considered malicious or risky in future detection. This change is made to the policy that we were using, which was used for all our clients (because it was the most restrictive policy possible, since we had every detection method supported by Ming Yan enabled), meaning that it will apply to all the clients that we were using.


事件处理

* 处理状态： 确认为事件 [✓ 确认为非事件]

☐ 加入该主机所在策略的可信名单中

可信名单的内容只产生普通日志，不会产生告警事件

Figure 14: The process of adding the process to the whitelist.

8 Results

After making the changes to the detection policy mentioned in the previous section, we found that this false positive was no longer appearing upon repeating the steps that we used to generate it the first time (and that regularly caused it whenever we tried). While it is not entirely clear why Ming Yan was classifying accessing the web page from the same host as potential reverse shell behavior, but not when it was being accessed in the same way from another host on the network, this process clearly should not be flagged as malicious.

Unfortunately, fixing the “Malicious IP” false positive is outside the scope of this project. We consulted the Ming Yan developers about this, who gave us two options for how this could be fixed: 1) disable detection of malicious IP addresses in the policy configuration, or 2) modify the server’s SQLite database of IP addresses to change this address to not be recorded as malicious. It would be practical to detect if someone using this machine were connecting to an IP address that was malicious; however, our partner’s GitHub homepage clearly is not. We believe that turning off malicious IP detection is not a good solution but cannot make changes to the server’s database ourselves. As such, this remains a false positive, and we are unsure what other IP addresses might be registered as “malicious” that should not be.

9 Future Work

We believe that there is room for more investigation regarding how serious the generation of false positives becomes with the use of the modified client that enables better detection within containers. A possible course of action would be to plan out and structure more complicated application structures that may run using Docker containers on these hosts. Since we now know that Nginx, Impala, and Redis have been cited by the developers as common causes of false positives in the past, it would be good to work with these services to investigate complex workloads more thoroughly to isolate the commands or configurations that cause the false positives on the modified client. Once it has been better investigated what generates these false positives, it would be possible to begin making modifications to the source code attempting to find the places where these processes are being flagged and see why they are being flagged.

We also suggest that research be done into the classification of IP addresses as malicious. Until the database is filtered for IP addresses that are not truly malicious, this false positive will continue to appear.

10 Conclusion

While there is still work to be done on this project, we believe that the signs we have seen so far point to Ming Yan being effective at detecting attacks within Docker containers, especially when using the modified client, which does not actively try to ignore behaviors that occur within Docker containers. While using the modified source code client, Ming Yan was able to detect all the malicious behavior and attacks that we tested on it, but it generated some false positives in the process. Based on our examination of the false positive generation, it is more of a boon than a bane to allow Ming Yan to detect attacks within containers, because we observed very few false positives, and the primary false positives that we observed were easily fixed from within the server interface so that they do not appear in the future. However, the ramifications of the changes that we made have not been fully studied; it is unclear if whitelisting docker-proxy could lead to an undetected attack later. A more in-depth study using more complicated container workloads is necessary to fully understand Ming Yan’s capabilities and the possible trade-offs of enabling detection within containers.

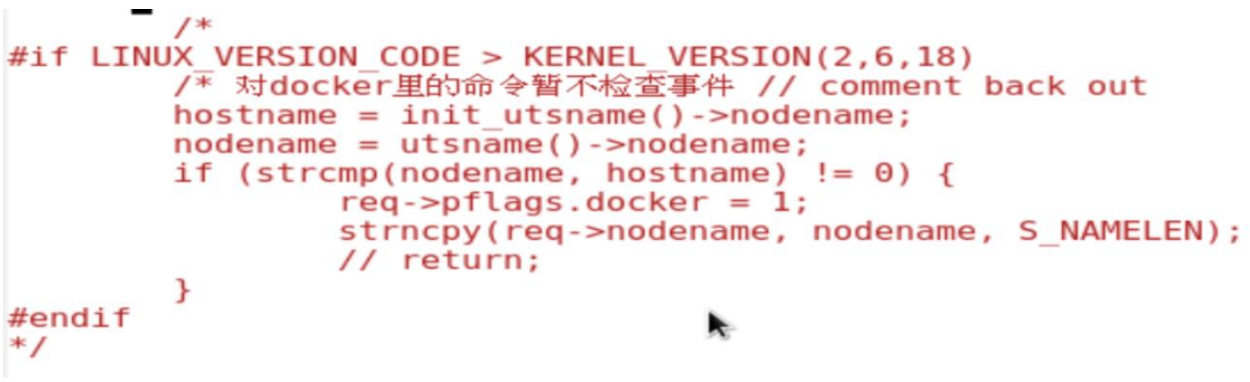
Acknowledgements

We would like to thank Dr. Yan Chen, Xiao Luo, and Ling Jin for their guidance with this project.

Appendix 1 Code and Policy Changes

1) Code modifications

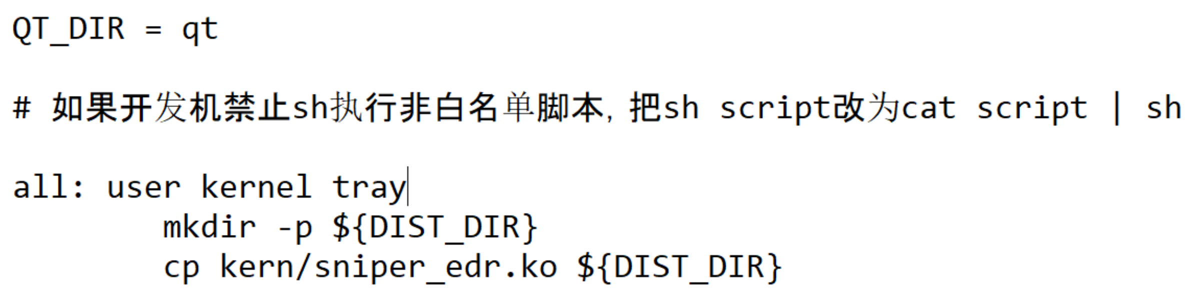
The change we made in the source code is from the /kern/execve_hook.c file. Below is a screenshot of the snippet of code that we removed; in the original version, this code was not commented out.



```
        /*
#if LINUX_VERSION_CODE > KERNEL_VERSION(2,6,18)
        /* 对docker里的命令暂不检查事件 // comment back out
        hostname = init_utsname()->nodename;
        nodename = utsname()->nodename;
        if (strcmp(nodename, hostname) != 0) {
                req->pflags.docker = 1;
                strncpy(req->nodename, nodename, S_NAMELEN);
                // return;
        }
#endif
*/
```

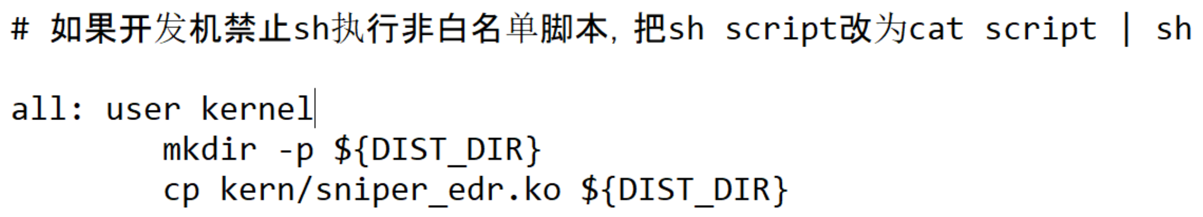
Figure 15: The block of code we removed from /execve_hook.c

A copy of the modified /kern/execve_hook.c file was sent to our mentors with this report. We also modified a line in the Makefile for the project from “all: user kernel tray” (below, fig. 16) to “all: user kernel” (fig. 17). We also include linux-client-src/Makefile in our deliverable to our mentors.



```
QT_DIR = qt

# 如果开发机禁止sh执行非白名单脚本，把sh script改为cat script | sh

all: user kernel tray
        mkdir -p ${DIST_DIR}
        cp kern/sniper_edr.ko ${DIST_DIR}
```

Figure 16: The original line in the Makefile.

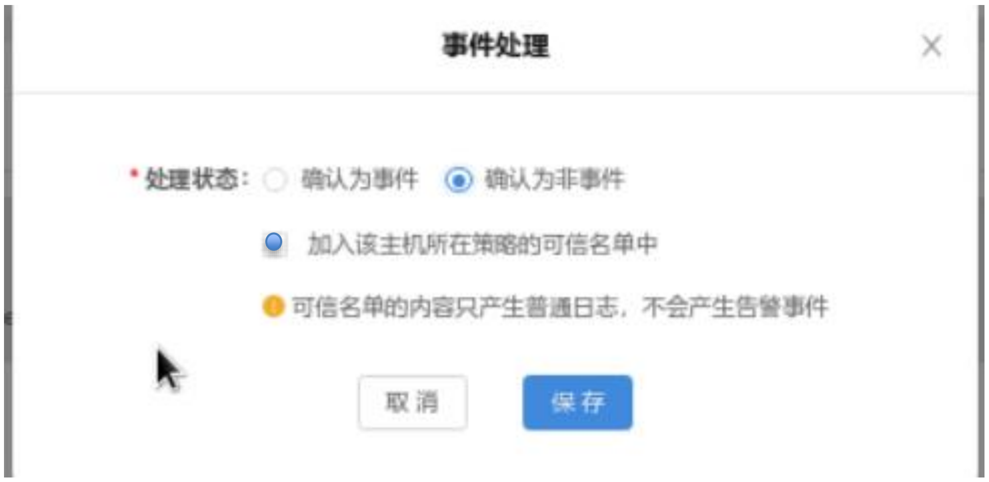


```
# 如果开发机禁止sh执行非白名单脚本，把sh script改为cat script | sh

all: user kernel
        mkdir -p ${DIST_DIR}
        cp kern/sniper_edr.ko ${DIST_DIR}
```

Figure 17: The modified line in the Makefile.

2) Policy modifications

The policy change we made to resolve the false positives was to add the docker-proxy process to the whitelist on the Ming Yan server, as shown in fig. 18. This policy change has to be added manually to the Ming Yan server configuration on top of the maximum-security policy (e.g., every detection mechanism enabled). The policy we used did not enable any blocking of malicious behaviors; the update below should work even when a user’s policy does enable blocking.



* 处理状态： 确认为事件 [✓ 确认为非事件]

[✓ 加入该主机所在策略的可信名单中]

可信名单的内容只产生普通日志，不会产生告警事件

Figure 18: Whitelisting the docker-proxy process.