Enhancing Android Debug Bridge (ADB) Security: A High-Interaction Honeypot Approach

Vissarion Moutafis
TU Delft
Delft, Netherlands
v.moutafis@student.tudelft.nl

Velyan Kolev
TU Delft
Delft, Netherlands
v.p.kolev@student.tudelft.nl

Tsvetomir Hristov
TU Delft
Delft, Netherlands
t.hristov@student.tudelft.nl

Dea Llazo
TU Delft
Delft, Netherlands
d.llazo@student.tudelft.nl

ABSTRACT

In the realm of smartphone technology, Android holds a dominant position with over 70% market share worldwide. However, with convenience comes vulnerability, particularly in the form of the Android Debug Bridge (ADB), a tool that enables seamless communication between Android devices and computers. Exploitation of ADB vulnerabilities poses significant cybersecurity risks, necessitating proactive measures to safeguard Android devices. To address this, we present a novel approach using a high-interaction honeypot deployed within a virtual machine environment. Leveraging Docker containers to emulate Android devices, our honeypot intercepts and analyzes ADB traffic to capture real-world attack scenarios. Our infrastructure offers ease of deployment, scalability, and effectiveness in detecting and analyzing ADB threats. Future enhancements aim to broaden protocol coverage and refine emulation environments, advancing cybersecurity defence in the Android ecosystem.

1 INTRODUCTION

Android is the leading smartphone operating system used in the world with over 70% market share [13]. As Android is open-source, Google provides an easy way for developers to test their applications both wired using a USB connection, as well as wireless over Wi-Fi. This software is called Android Debug Bridge (ADB) and allows developers to communicate with an Android device or emulator from a computer. It facilitates various tasks such as installing and debugging apps, accessing the device's shell, transferring files between the device and the computer, and forwarding ports for debugging purposes.

When wireless debugging is enabled on an Android device, a computer can connect to it over Wi-Fi using an ADB server instance which scans the network on some default ports. While this feature is indispensable for developers, it inadvertently allows for exploitation. Malicious actors can set up their ADB servers to scan for vulnerable devices and gain unauthorized access, which poses serious cybersecurity risks.

Exploitation through ADB has been exemplified by numerous Android attacks that utilize ADB to inject malicious code onto vulnerable devices [9] [6]. These exploits emphasize the urgency of addressing the security implications associated with ADB usage, thus motivating the objective of our project.

Honeypots have long been recognized as effective tools for capturing and analyzing attacker behaviour as they provide a simulated environment for attackers to interact with. Several honeypots have been developed to detect and deter malicious activity for ADB [5]. Observing scans aimed at ADB shows how common these security risks are [14].

To contribute to the ongoing efforts in securing ADB [1], we have developed a high-interaction honeypot deployed within a virtual machine (VM) environment, utilizing Docker containers to simulate Android devices. By leveraging official Google images of Android devices, we can emulate device behaviour without the need to implement the ADB protocol explicitly. This setup allows us to intercept and analyze all traffic exchanged via ADB between potential attackers and the emulated Android devices.

To enhance the honeypot's effectiveness, we implement measures to assign the same emulator to attackers identified by their IP addresses. This ensures continuity in monitoring attacker behaviour and allows us to track changes made over time. Additionally, periodic resetting of emulated devices helps mitigate the impact of malicious activities on the VM, maintaining the integrity of the honeypot environment.

In summary, our project aims to strike a balance between realism and security in the design of an interactive ADB honeypot. By capturing real-life attack scenarios in a controlled environment, we seek to gain valuable insights into ADB-related vulnerabilities and enhance the resilience of Android devices against malicious exploitation.

2 BACKGROUND AND RELATED WORK

In this section, we provide background knowledge necessary for understanding the problem outlined in the introduction and the technologies used in our infrastructure. Furthermore, this section expands on the state-of-the-art and current solutions for detecting and analyzing ADB malicious traffic, including existing honeypots and other relevant tools. By examining these solutions, we highlight the importance of our approach and its novelty.

Honeypots. Honeypots represent a cybersecurity strategy employed to detect and study malicious actor behaviour. These decoy systems are purposefully designed to attract attackers and gather intelligence on their approach, tools, and motives. By emulating

vulnerable systems or services, honeypots provide a controlled environment for observing attacker behaviour without risking the integrity of production systems.

There are several types of honeypots, categorized based on their level of interaction and deployment. High-interaction honeypots offer a fully functional environment that closely mirrors a legitimate system, enabling extensive interaction with potential attackers. In contrast, low-interaction honeypots provide limited functionality, emulating only specific services or parts of protocols.

In the context of securing ADB, honeypots serve as invaluable tools for monitoring and analyzing ADB-related traffic. The main purpose of these honeypots is to support the progress of ADB security research, aiming to improve the possible defence mechanisms that can be used on Android devices against malicious attacks. By deploying realist honeypots that emulate ADB-enabled devices, we can observe how the attack surface changes over time and create appropriate countermeasures.

Android Debug Bridge Protocol. Used across a diverse range of devices, including smartphones, tablets, and IoT devices running the Android operating system, ADB enables developers to install, debug, and manage applications directly on connected devices or emulators. Android Debug Bridge (ADB) is a versatile command-line tool that lets you communicate with an Android device. ADB acts as a communication interface by providing access to a Unix shell that you can use to run a variety of commands on a device for mostly debugging and Android development purposes. With commands supporting functions such as application installation, debugging, shell access, file transfer, and port forwarding, ADB plays a central role in application development workflows.

However, its default configuration, operating over TCP/IP on port 5555, poses security risks, including unauthorized access, malicious app installations, and data leakage. To mitigate these risks, best practices include securing ADB access by enhancing the integrity of the ADB protocol, implementing network segmentation, and ensuring regular security updates [9]. Understanding the ADB protocol is essential for safeguarding Android development environments against potential threats and vulnerabilities.

Docker. Docker is a powerful tool used in software development and cybersecurity. It employs containerization technology, allowing applications to run in isolated environments known as containers[7]. Containers package the application with all its dependencies, ensuring consistent and efficient execution across different platforms.

One key benefit of Docker is its security features. By isolating applications within containers, it prevents breaches from spreading to the underlying system, making it challenging for attackers to compromise sensitive data [8]. Leveraging Docker, security researchers can construct realistic, lightweight, and compatible honeypot environments that emulate different systems and protocols, facilitating the detection and analysis of potential threats.

Reverse Proxies. Reverse proxies serve as intermediaries between clients and servers, facilitating the efficient management of network traffic. Unlike traditional forward proxies that primarily handle requests from clients seeking access to external resources, reverse proxies operate on behalf of servers, intercepting incoming client requests and directing them to the appropriate services based on predefined rules or criteria. By abstracting the underlying server infrastructure from clients, reverse proxies enhance scalability, reliability, and security, while also simplifying maintenance and troubleshooting tasks for network administrators.

In this project, we use NGINX [11] as a reverse proxy that forwards the connections to the emulators and also acts as a load-balancer. NGINX is a high-performance web server and proxy server known for its stability, robustness, and efficiency in handling high volumes of traffic. It effectively routes incoming requests to the appropriate services, optimizing resource utilization and ensuring seamless operation of the honeypot infrastructure.

IPS. An Intrusion Prevention System (IPS) is a specialized system, designed to identify probable intrusions and prevent them, in real-time. IPS systems can be implemented both in software and hardware and usually stay hidden from the end hosts of the communication. Although they are not the main line of defence, IPSs are used to enhance security, as they stay hidden from the rest of the network, offer high inspection capabilities, and real-time traffic inspection and prevention. Our approach employs an IPS to be able to monitor and contain all traffic on the system.

Related Work. In this subsection, we will dive into existing ADB Honeypot solutions, ADB traffic analyzing tools, and other similar technologies. Additionally, we'll highlight the enhancements and unique features we introduce to the state of the art.

ADBHoney. ADBHoney[5] is a low-interaction honeypot specifically designed for Android Debug Bridge (ADB) over TCP/IP. It aims to emulate ADB services over port 5555, enabling commands like `adb connect`, `adb push`, and `adb shell`. The honeypot redirects data and files to stdout and disk, respectively, capturing basic interactions with the emulated ADB service by using default responses to a set list of commands. While it provides valuable insights into potential threats targeting devices with exposed port 5555, it lacks advanced features such as sophisticated response mechanisms or deep analysis of captured data. Building upon this work, ADB Honeypot is a Twisted implementation of an ADB honeypot, focusing on monitoring and analyzing ADB-related threats [2]. While it offers enhanced functionality compared to ADBHoney, such as the ability to distinguish between different implementations and handle various commands, it still primarily operates as a low-interaction honeypot.

What sets our solution apart from existing ADB honeypots is its commitment to offering both realism and security. Inspired by ADBHoney and ADBhoneypot, we introduce novel features and enhancements. A key focus is on providing a consistent highly interactive experience for each attacker by emulating the ADB protocol on a new infrastructure that helps provide realistic interaction while maintaining security.

Android Malware Detection through Network Traffic Analysis. This approach[3] focuses on detecting Android malware through network traffic analysis. By comparing the network traffic of malware with benign apps, distinguishing features are identified and used to build a decision tree classifier. The workflow involves data collection using tcpdump, feature extraction and labelling, and training/testing of a machine learning model. This method offers

a proactive approach to detecting Android malware by analyzing network traffic patterns and behaviours.

Such tools could be used on top of our infrastructure either to already detect suspicious traffic or to further train the model based on the analysis and the data captured by our honeypot. Similar analysis techniques will be useful to analyze the traffic that we will capture on our infrastructure.

T-Pot. T-Pot[15] is a comprehensive honeypot system that incorporates various honeypots, including ADBHoney, along with additional tools for network monitoring and analysis leveraging Docker to deploy its infrastructure. It provides a unified platform for cybersecurity research and threat intelligence gathering with features such as real-time performance monitoring and data visualization, making it a versatile tool for detecting and analyzing malicious activities.

Given its utility, our infrastructure could seamlessly integrate into the platform. Leveraging Docker images, the process of deploying our honeypot on this platform could be streamlined, thereby simplifying the data analysis process and adding value to our infrastructure.

Dockerpot. Dockerpot[10] is a docker-based solution designed to create and deploy honeypots using Docker containers. It provides a modular and scalable approach to honeypot deployment, allowing users to easily spin up multiple honeypots on the same host. Dockerpot includes features such as automated cleanup, service configuration, and logging, making it suitable for various types of honeypots. It offers flexibility in customization and configuration, allowing users to adapt the honeypot environment to their specific needs.

This solution served as our inspiration, prompting us to build upon its foundation. We were drawn to the concept of a customizable dockerized honeypot capable of implementing multiple protocols, thus leading us to develop an enhanced highly interactive infrastructure based on this idea.

3 METHODOLOGY

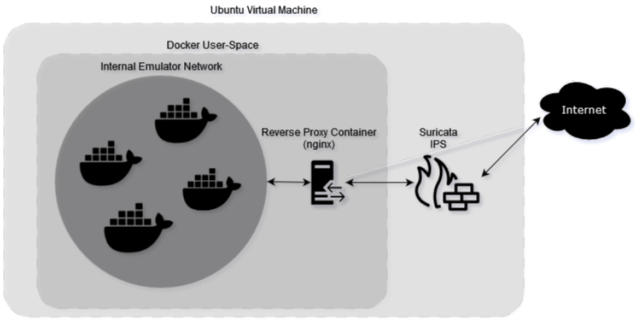

Figure 1: Overview of the system architecture

Architecture Overview. Our solution establishes three services inside a Virtual Machine (VM). The VM is used to easily be able to run the honeypots on external servers as well as to further separate any malicious behaviour from the host machine. The three services we use are an Intrusion Prevention System (IPS), a reverse proxy, and Android emulators.

Figure 1 shows an overview of the architecture. The goal of the architecture is to allow us to expose several Android machines (emulators) to any outside traffic and capture attempts made to them through Android Debug Bridge (ADB). As the emulators do not start any ADB communication by themselves, we assume that any traffic targeted towards ADB (port 5555) is suspicious and hence should be captured.

In order to achieve this goal, we spawn Android emulators inside Docker containers that are isolated from the internet using an internal network. Then, the reverse proxy is responsible for forwarding any traffic targeted towards port 5555 to one of these emulators. All the traffic is captured using the IPS as it sits inside the VM and hence can see any traffic even unrelated to ADB.

Virtual Machine (VM). The full infrastructure is deployed inside a Virtual Machine. This is firstly done so that in the slim chance that an attacker is able to break out of a Docker container, they are presented with a VM instead of an actual host machine. This further limits the attack surface and contains any damages done by attackers. Hence, the VM serves as a final line of defence against malicious parties. Additionally, having a VM enables us to run our solution on existing servers which allows for easier scalability on the data collection side. The VM allows for a sandbox where the Android emulators can be run.

Android Emulators. The backbone of this project depends on having an ADB emulator that is able to reproduce how the ADB protocol works so that attackers are fooled into connecting with it. As we wanted to make a high-interaction honeypot we opted for the approach of emulating an Android device directly which makes the ability of ADB connection trivial. The emulated devices are run inside Docker containers so we can spawn multiple copies of them for scalability of the system. We use Google's Android emulator images sourced from their publicly available directory[4] as they provide a reliable and standardized platform that can be deployed easily.

Using Android emulators allows for any ADB commands to work out of the box. However, this poses a problem where now we need to separate them from the outside world to not spread malware and affect any other devices on the internet. To mitigate these risks and associated ethical implications we use an internal Docker network[7]. This network configuration effectively isolates the emulated devices from external connections and now they can only be contacted from other processes inside the network. Additionally, to maintain the stability and integrity of the emulation environment, we implemented a restarter Docker container with access to the VM's Docker socket. This container restarts the emulator containers at regular intervals, ensuring continuous operation and resilience against potential disruptions or failures. Furthermore, this behaviour limits rogue containers and decreases the chances of any break-out from the containerized environment.

In conclusion, to achieve the goal of simulating an ADB-capable device we emulate real Android devices that follow the ADB protocol explicitly. The emulators are enclosed in an internal network to limit their effect on the internet, however, that presents a new challenge that now we need a service that has access to the internal

network and can forward requests for ADB to these devices. Hence, why we use a reverse proxy.

Reverse Proxy. The reverse proxy serves as a middleman between the Android emulators and the VM. It has two interfaces - one connected to the internal Docker network where the emulators reside, and one connecting to the outside. The main purpose of the reverse proxy service is to forward any traffic aimed at port 5555 (default ADB port) to the emulator's internal network. We use Nginx to serve as our reverse proxy, more specifically, an Nginx reverse proxy Docker container[1] as it can efficiently manage incoming traffic.

Additionally, as we want the system to be scalable and allow attackers to have a consistent view of an Android device, we spawn multiple emulators. Then, we hash the source IP of the attacker and map it to one of the emulators. This makes it so that an attacker that sends multiple requests through the same IP will always be given the same emulator and his changes will persist (until the emulator is restarted). Leveraging request-IP address hashing, the reverse proxy allows for equal distribution of incoming requests among the connected emulators, ensuring optimal resource utilization and load balancing.

By serving as a centralized gateway for incoming traffic, the Nginx reverse proxy container enhances the scalability, reliability, and security of our honeypot deployment, while also facilitating efficient management and redirection of network traffic. One thing it does not do is record the traffic that is sent to the containers, for that, we use an IPS.

Intrusion Prevention System (IPS). We use an Intrusion Prevention System (IPS) to monitor all traffic that goes into the VM. This allows us to block certain traffic that should not be allowed, as well as, log all the traffic we need. The IPS is set up on the VM (and not in a Docker container) as we need to oversee all traffic. If it was set up inside a Docker container, or in the reverse proxy container, it would not see outgoing traffic that is not related to ADB. This is important to capture to be sure that we know all the traffic that is coming or going to the emulators.

We use Suricata[12] for our IPS as it offers IPS capabilities coupled with flexible deep packet inspection based on content-specific rules. It is a flexible and user-friendly way to establish a robust defence mechanism capable of detecting and stopping malicious activities within our network environment. Currently, Suricata is configured to permit all incoming and outgoing traffic, while logging alerts for any attempts to execute ADB commands within the honeypots. The current infrastructure is specifically designed to allow users to add new rules to block or alert specific traffic. This can be used for example if you want the honeypot to block a specific attack on the ADB protocol.

Throughout our experimentation phase, we explored various strategies to integrate Suricata within our deployment. Initially, we attempted to integrate Suricata and Nginx on the same service/-container. However, this configuration rendered outbound traffic from the Android devices, unrelated to the ADB protocol, invisible to Suricata, thus impeding its effectiveness. Subsequently, we tried to deploy Suricata within a separate container, serving as a gateway for all network traffic. However, inherent limitations in Docker's network configuration made it impossible to override the default gateway, thereby impeding the desired traffic redirection. Ultimately, we devised a solution wherein Suricata operates as an IPS on both the internal and bridge network interfaces at the VM user level. This approach not only overcame the aforementioned problems but also ensured the seamless integration of Suricata into our honeypot infrastructure, as explained in the Methodology section.

To enable Suricata's functionality, we leveraged an `NFQUEUE` configuration of `iptables` together with a `systemd` service. Through this configuration, all traffic from both Docker network interfaces was forwarded to an `NFQUEUE`, enabling Suricata to inspect and enforce IPS rules based on user-defined criteria. Additionally, we implemented a `systemd` process to ensure the Suricata service initiates upon VM startup, thereby guaranteeing deployment persistence across system restarts or crashes.

Flexibility of the deployment. Our deployment framework offers great extensibility, providing a generic out-of-the-box honeypot solution that supports any type and quantity of honeypots packaged within Docker images. Leveraging Docker containers, users can effortlessly deploy and scale their honeypot environments to suit specific requirements, while both the Nginx reverse proxy and Suricata components are configurable to accommodate diverse experimentation contexts running on specific ports. See Appendix A for follow-up instructions on how to add a new protocol as a honeypot. Nginx's customizable redirection capabilities and Suricata's flexible IPS rules and traffic inspection parameters[12] allow users to adjust the technical deployment configurations and support various experiments. This deployment was inspired by the T-Pot implementation[15] which consists of multiple honeypots for various protocols. T-Pot combines Suricata with the Elastik stack in order to visualize reports and uses its own load-balancing mechanism. The whole product is delivered as an ISO which can be installed in a virtual machine and is highly configurable.

4 RESULTS

In this section, we show the output and behaviour of the system when it is set up. As previously mentioned, the system runs on a Virtual Machine where three services are set up - Android emulators, a reverse proxy (NGINX), and an IPS (Suricat).

First, we tested that outside entities can contact the containers on port 5555. Listing 1 shows the configuration of the reverse proxy where we can see that it has IP "192.168.21.1" as its public IP and "192.168.21.2" as its private IP. Figure 2 shows an attempt from the VM to access the emulators and run an ADB command.

As the VM uses the outside interface it can only communicate with the containers through the reverse proxy. Hence, in the logs, we see that when the `adb shell ls` command is run, the reverse proxy has forwarded the requests through one of its ports on the public network (192.168.21.1:33878) to the 5555 port on its internal one (192.168.21.2:5555). Hence, allowing other hosts that are outside of the internal Docker network to connect to the emulators. Additionally, we look inside the emulators and try to contact the external IP of the reverse proxy as well as a public Google server. Figure 3 shows that in both cases we cannot reach the host as the

[1] https://hub.docker.com/_/nginx

```
nginx:
container_name: rev-proxy
image: nginx:latest
ports:
  - 5555:5555
networks:
  br-public:
    ipv4_address: 192.168.21.2
  br-android:
    ipv4_address: 192.168.22.2
environment:
  - NGINX_ENTRYPOINT_QUIET_LOGS=1
  - NGINX_REAL_IP_HEADER=proxy_protocol
  - NGINX_REAL_IP_RECURSIVE=on
volumes:
  - ./nginx.conf:/etc/nginx/nginx.conf:ro
restart: always
```

Listing 1: Reverse Proxy Service

internal network does not allow the emulators to access the outside world. Thus, we achieve what we initially set out to do - allow hosts to only connect on port 5555 in the emulators through the reverse proxy, and block all outgoing traffic in the emulators.

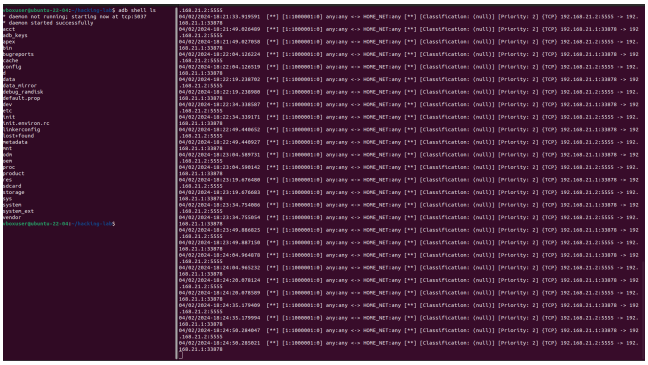

Figure 2: Running ADB from outside the network and monitoring suricata's "fast.log". Notice how the ADB client (192.168.21.1) connects through the reverse proxy in address 192.168.21.2:5555.

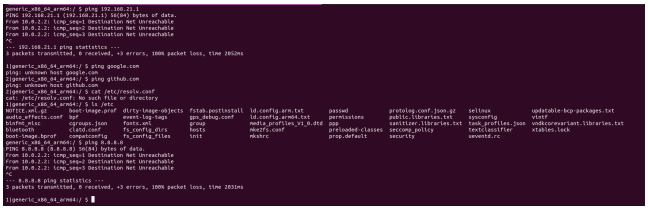

Figure 3: Emulator shell cannot reach addresses outside the internal network, i.e physical host on public bridge, 192.168.21.1, or public google DNS server 8.8.8.8

Finally, we should mention that currently, we keep track of two types of logs where one is used to store basic alert data (Figure 2) and in the other, we store the packet information of ADB requests (Listing 2). For the latter, there are two types of payload saved - byte payload (base64 encoding denoted by `payload` keyword) as well as printable payload (ASCII payload denoted by `payload_printable` keyword). This is because the base64 encoding allows for faster parsing of dangerous user commands, without necessarily parsing the ASCII payload. The ASCII payload is useful as it provides a human-readable representation of the original payload.


```
{
   "timestamp":"2024-04-02T20:04:58.215249+0200",
   "flow_id":2046813460337642,
   "event_type":"alert",
   "src_ip":"192.168.21.1",
   "src_port":53882,
   "dest_ip":"192.168.21.2",
   "dest_port":5555,
   "proto":"TCP",
   "pkt_src":"wire/pcap",
   "alert":{
      "action":"allowed",
      "gid":1,
      "signature_id":1000001,
      "rev":0,
      "signature":"any:any <-> HOME_NET:any",
      "category":"",
      "severity":2
   },
   "app_proto":"failed",
   "direction":"to_server",
   "flow":{
      "pkts_toserver":5,
      "pkts_toclient":3,
      "bytes_toserver":451,
      "bytes_toclient":460,
      "start":"2024-04-02T20:04:55.214416+0200",
      "src_ip":"192.168.21.1",
      "dest_ip":"192.168.21.2",
      "src_port":53882,
      "dest_port":5555
   },
   "payload":"T1BFTgIAAAAAAAAAJAAAAAAAAACwr7qxc2hlbGws
        djIsVEVSTT14dGVybS0yNTZjb2xvcixyYXc6bHMA",
   "payload_printable":"OPEN........$..........shell,v2,
    TERM=xterm-256color,raw:ls.",
   "stream":0
}
```


Listing 2: Captured ADB traffic where attacker tried to run `ADB shell ls`

5 DISCUSSION

Our current implementation allows users to set up a Virtual Machine running several services that expose two Android emulators on port 5555. We have a reverse proxy service that forwards requests made on port 5555 to one of these emulators such that the same IP gets the same emulator each time. Additionally, we monitor all traffic using an Intrusion Prevention System which can be configured to block specific traffic if needed. Finally, the Android emulators are configured in a Docker internal network to prevent rogue emulators from targeting other users within the network.

The infrastructure is easy to set up and configure. It offers a new high-interactive approach to capture attempts at exploiting the

Android Debug Bridge without sacrificing the functionality of the protocol. Additionally, we place focus on the effects of deploying such a realistic honeypot by isolating and containing the attacker's activity limiting the amount of damage they can cause. The current state of the infrastructure allows for changing the service that is analyzed by switching the emulator containers and the ports that the reverse proxy inspects. Our honeypot builds upon previous solutions relating to ADB as well as substituting protocols.

Several limitations and improvements can be pointed out. First, the internal network currently blocks all outgoing traffic. The decision to block all outbound traffic aims to restrict the freedom of potential attackers since attacker behaviour is limited by the offline setting of the compromised device. If at a later stage, it is needed that emulators communicate to the internet, i.e. to download binaries, the internal network should be disabled and new Suricata rules should be set up to limit the outbound traffic. However, one issue that arises is the ethical concerns of running machines that are made to be compromised and their resources can then be used for other malicious behaviour. Hence, we have decided to make the emulators block all outgoing traffic through the internal network which mitigates these concerns.

Another challenge is log reconstruction post-execution of the `adb shell` commands arises, as each typed letter on the command-line interface is continuously transmitted to the machine, complicating log reconstruction and threat analysis. This can be fixed by either blocking the *adb shell* command and allowing only the `adb shell {parameter}` commands. Otherwise, once a user connects using `adb shell` their input is sent in TCP packets one by one limiting the blocking capabilities of Suricata. Another way to mitigate this is to somehow reconstruct the full commands based on the single letters sent after the `adb shell` command.

Currently, a straightforward rule set has been implemented to alert on inbound and outbound traffic relating to the ADB protocol. However, as some attacks might prompt specific behaviour of the devices that happens internally, implementing in-device logging to monitor rogue processes could provide deeper insights into potential attacks, particularly concerning the execution of arbitrary executable files. Finally, an improvement that can be made is to construct the docker-compose when starting the system and allow the VM user to choose how many emulators to spawn.

6 CONCLUSION

In this project, we addressed the pressing need for robust and adaptable honeypot infrastructure to detect and analyze malicious Android Debug Bridge (ADB) traffic effectively. The increase in ADB-based attacks in the last years highlights the importance of proactive measures to safeguard Android devices against exploitation.

Our solution introduces an innovative infrastructure capable of not only detecting ADB threats but also serving as a versatile framework, modifiable for a variety of protocols. By leveraging Docker, a reverse proxy, an Intrusion Prevention System (IPS), a Virtual Machine, and Android emulators, we have constructed a highly secure and scalable environment capable of an in-depth analysis of ADB traffic.

Our infrastructure is easy to deploy, thanks to a streamlined setup process while ensuring detailed configuration options for tailored honeypot environments. Moreover, our solution's incorporation of multiple emulators, enhances the authenticity of captured interactions, contributing to more accurate threat analysis of real-world scenarios.

One of the key strengths of our honeypot infrastructure lies in its proactive approach to security, demonstrated by the strict control over outgoing traffic to prevent the spread of malware to genuine devices and the use of an IPS. This not only protects the broader Android ecosystem but also mitigates potential risks associated with honeypot deployment.

Looking ahead, future work could explore expanding our infrastructure to encompass additional protocols beyond ADB, broadening its utility in detecting and analyzing diverse cyber threats. Additionally, continued refinement of the emulation environment and response mechanisms, based on the data analysis of the traffic we get out of our infrastructure, could further enhance the realism and effectiveness of our honeypot solution. Overall, our project represents a significant step forward in ADB security research, offering a comprehensive and adaptable framework for combating emerging threats in the Android ecosystem.

ACKNOWLEDGMENTS

This project was made under the supervision of Harm Griffioen (Cyber Security Group, Delft University of Technology).

REFERENCES

[1] Last updated November 8 and Alan Bavosa. 2023. How to block Android Debug Bridge (ADB) exploits, protect Android apps. https://www.appdome.com/how-to/mobile-fraud-detection/synthetic-fraud-detection/block-android-debug-bridge-adb-exploits-protect-android-apps/
[2] Bontchev. 2023. GitHub - bontchev/adbhoneypot: An Android Debug Bridge honeypot. https://github.com/bontchev/adbhoneypot
[3] Devu. 2020. GitHub - devu-62442/Android-Malware-Analysis. https://github.com/devu-62442/Android-Malware-Analysis
[4] google. [n. d.]. GitHub - google/android-emulator-container-scripts. https://github.com/google/android-emulator-container-scripts
[5] Huuck. [n. d.]. GitHub - huuck/ADBHoney: Low interaction honeypot designed for Android Debug Bridge over TCP/IP. https://github.com/huuck/ADBHoney
[6] Sungjae Hwang, Sungho Lee, Yong-Dae Kim, and Sukyoung Ryu. 2015. Bittersweet ADB. *Bittersweet ADB: Attacks and Defenses* (4 2015). https://doi.org/10.1145/2714576.2714638
[7] Docker Inc. 2024. "Home". https://docs.docker.com/
[8] Docker Inc. 2024. "How does it work?". https://docs.docker.com/desktop/hardened-desktop/enhanced-container-isolation/how-eci-works/
[9] Author links open overlay panelKrzysztof Opasiak a b, a, b, AbstractUniversal Serial Bus (USB) is currently one of the most popular standards that controls communication between personal computers (PCs), their peripheral devices. Thus, D. Barral, K. Cabaj, B. Camredon, and M.O. Dominic Spill. 2018. (in)secure android debugging: Security analysis and lessons learned. https://www.sciencedirect.com/science/article/pii/S016740481831023X
[10] Mrschyte. [n. d.]. GitHub - mrschyte/dockerpot: A docker based honeypot. https://github.com/mrschyte/dockerpot
[11] Inc. NGINX. 2024. Advanced Load Balancer, Web Server, & Reverse Proxy - NGINX. https://www.nginx.com/
[12] Open Information Security Foundation (OISF). 2023. Documentation - Suricata. https://suricata.io/documentation/
[13] Ahmed Sherif. [n. d.]. *Mobile OS market share worldwide 2009-2023.* https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/
[14] Pedro Tavares. 2018. ADB shell attacks on the rise. https://seguranca-informatica.pt/adb-shell-attacks-on-the-rise-10th-of-july/
[15] Telekom-Security. [n. d.]. T-Pot - the all in one honeypot platform. https://github.com/telekom-security/tpotce

A ADDING MORE HONEYPOTS

The code and documentation for the honeypot framework is available in Github[2] where there are also notes on how to extend the honeypot infrastructure and add your own. The requirements to add a new protocol as a honeypot are as follows

- create a docker image of your honeypot service
- configure nginx reverse proxy to re-route the connections to your service
- if you want your service to restart in a regular period of time, adjust the restarter container configurations

You can see a template of a new service docker-compose service object in Listing 3.

```
new-hpot:
image: <DOCKER IMAGE>
container_name: new-hpot
hostname: new-hpot
environment:
  # add any environmental variables here
networks:
  # add the hpot to the internal network
  - br-internal
depends_on:
  # make sure to build and start nginx first
  - nginx
```

Listing 3: Template for a new honeypot service in compose.yaml

[2] https://github.com/VissaMoutafis/hacking-lab