Evaluating eBPF based tracing for analyzing packers on Android

Lab Presentation

Overview

- What is my topic?
- Background
  - Android, Packers, eBPF and BCC
- Implementation
  - Platform, Analysis Component
- Evaluation
- Conclusion

What is my topic?

- As the title already suggests, this Lab investigates the aptitude of eBPF based tracing of packers on Android
- To give an evaluation, an exemplary analysis platform was implemented upon which the aptitude, strengths and drawbacks of eBPF based tracing is evaluated
- Further a comparison to the popular tool `strace` is made

Background

- Android
- Packers
- eBPF and BCC

Background - Android

- Linux based open source OS with a focus on mobile devices
- Open source nature allows for highly customized versions
  - For example, versions with a focus on security research
- Android's Linux Kernel allows developers to also benefit from ongoing Kernel development
  - This includes functionalities such as eBPF

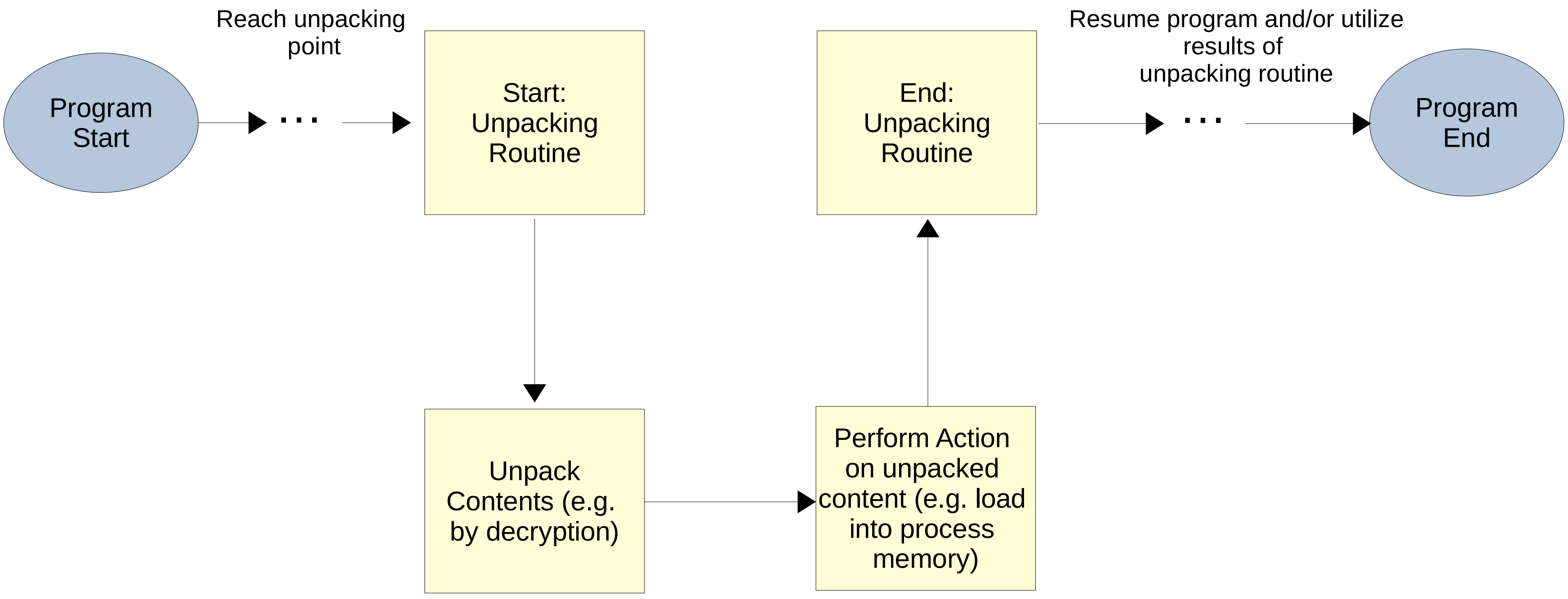
Background - Packers

- Simplified: a form of obfuscation aiming to protect sensitive content from being directly accessed used
- Malware authors make use of packers to evade detection of their payload by Antivirus engines (and/or to remain undetected longer)
- Packers also find use in legitimate scenarios, for example, to enforce Digital Rights Management (DRM)
- Focus in this Lab: Malicious apps and Dex/ELF contents

Background - Packers

- Mechanics of a packer (simplified):
  - At first, sensitive content (e.g. ELF lib) is brought into a protected format
  - Protected content shipped with/downloaded by the application
  - At a certain point in the application's runtime, an unpacking routine is called to restore the original version of the content for further processing (e.g. loading code into memory)

Background - Packers



Background – eBPF and BCC

- eBPF is short for *extended Berkley Packet Filter*
- Based on BPF, which is a mean to realize high performance packet filtering
- In general, BPF/eBPF is usually mentioned more in a performance measurement context than a security context
- In BPF a virtual machine runs within the OS kernel that performs a filtering task

Background – eBPF and BCC

- Userland programs can load filters (as bytecode) into that VM and will be provided with the respective output
- However, for stability and security not any arbitrary logic can be loaded into the VM
- A *Verifier* ensures, that the proposed bytecode fulfills certain characteristics
  - For example, no possibility for infinite loops or out ouf bound reads/writes

Background – eBPF and BCC

- eBPF is very similar to the original BPF, with the exception, that the VM is now able to access additional event sources
- This enables eBPF to process data not only on packets but also on, e.g. *kprobes* and *uprobes*
  - Thus enabling kernel/user space tracing with eBPF
  - For example, to trace system and/or library calls

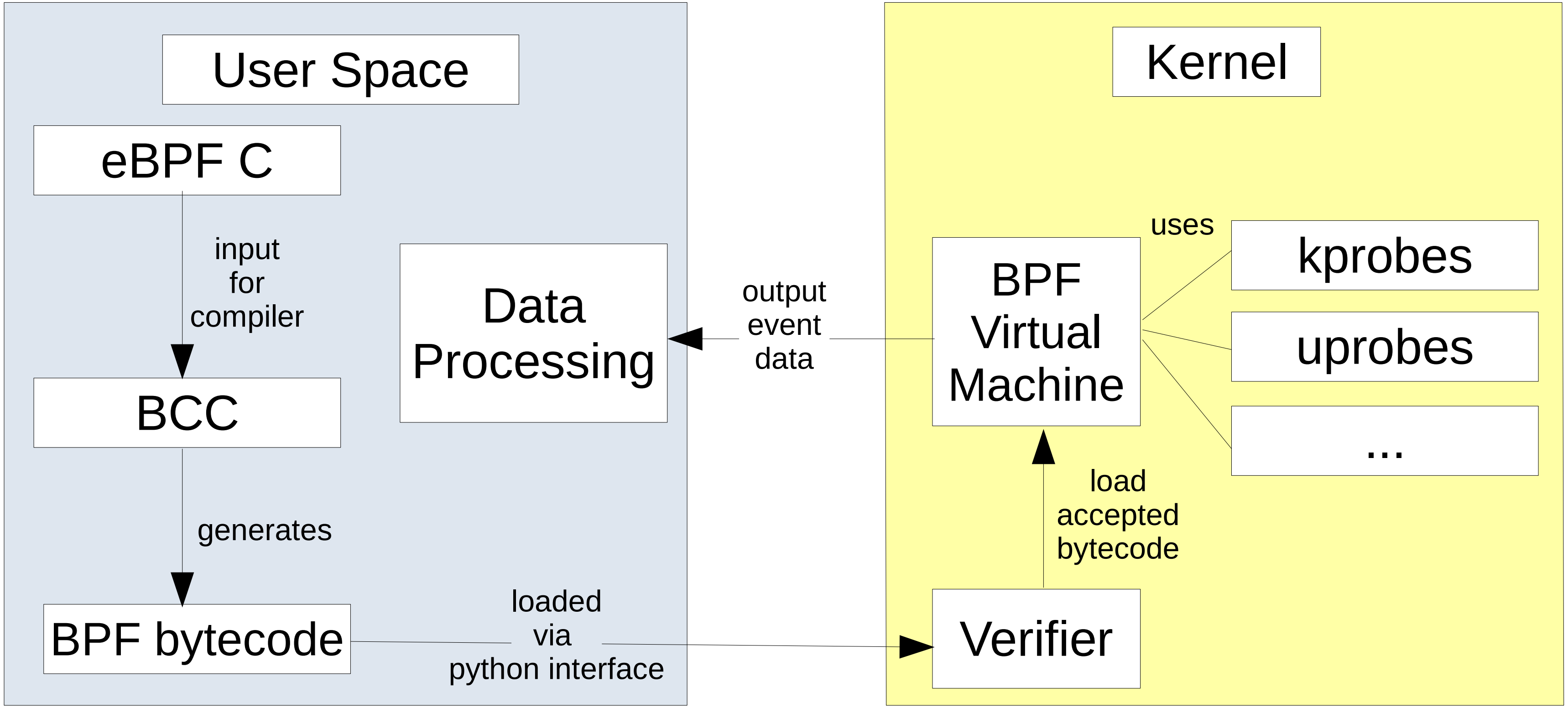
Background – eBPF and BCC

- However, implementing a meaningful logic in pure BPF VM bytecode is very difficult and error prone
- This is where BCC (BPF Compiler Collection) comes in
- BCC makes working with eBPF much easier as it:
  - Generates and loads bytecode into BPF VM
  - Provides simple access to the output data of the VM (via python interfaces)

Background – eBPF and BCC

- In BCC, the routines themselves are implemented in eBPF C, a restricted and limited C-type language
  - For example, lacks convenience functions (strlen, strstr)
  - Keep in mind, will run in the restricted/limited confinements of the BPF VM (no opening of files, etc.)
  - Also, the resulting bytecode after compilation still has to comply with the *Verifier*

Background – eBPF and BCC



Implementation

- **Building an Android platform with eBPF support**
  - Platform Requirements
  - Android Emulator, Cuttlefish, Android-x86
  - Remarks
- Realizing the Analysis Component
  - Installing BCC
  - Analysis Strategy
  - Implementing Analysis Logic

Implementation - Disclaimer

- The project was developed on a laptop with an i5-520 processor (4 threads) and 8GB of RAM
  - Therefore, performance was an issue, that influenced certain decisions

Implementation – Platform Requirements

- In order to support eBPF on Android, the kernel has to be compiled with an appropriate configuration
- Therefore, the target platform should either:
  - already possess an appropriate kernel
  - should make it possible to easily compile and run a customized kernel

Implementation – Platform Requirements

- The platform should also be x86-64 based to preserve performance
- Compatibility with as many applications as possible:
  - Running ARM based applications
  - Running older/newer applications
  - Running apps relying on Google APIs

Implementation – Building the Platform

- In this work, three projects were chosen as candidates
  - Android Emulator
  - Cuttlefish
  - Android-x86

- Due to the limited scope, alternatives like Genymotion and Corellium were not considered

Implementation – Android Emulator

- Google developed and catered towards application development
- Prebuilt images (>= 29) come with a kernel with eBPF support
- Prebuilt images are shipped with Google APIs installed
- Since Android 11 (API 30) ARM apps can run on x86-64 systems
  - Android 11 unfortunately “too new” for many sample apps
- Discarded due to incompatibilities, but great for getting first and easy experiences with eBPF on Android

Implementation – Cuttlefish

- Google's newest iteration for virtualizing hardware for Android
- Catered towards kernel hacking
- Had significant performance impact on system
- Is similar in the compatibility issues as Android Emulator
  - Therefore, also discarded
- However, very useful for exploring different kernel versions and how they affect eBPF

Implementation – Android-x86

- Open source project with the goal to run Android on x86 platforms
- Makes it possible to run on VirtualBox (better performance)
- Can support ARM based applications by using ARM-Translation (libhoudini)
- Unfortunately, the prebuilt images do not support eBPF in the kernels, so a customized Android had to be compiled
- Google APIs are also not available

Implementation – Android-x86

- An Android 7 (Nougat) debug build with a 4.9.194 kernel was chosen to be the target platform
  - Android 7 old/new enough to run most applications
  - Build variant allows for better system access (e.g. root access)
- For further compatibility, Google APIs were installed by using the open source project OpenGApps
- Time consuming to build, but the resulting platform is able to run a wider range of apps than the previous alternatives

Implementation – Analysis Component

- Building an Android platform with eBPF support
  - Platform Requirements
  - Android Emulator, Cuttlefish, Android-x86
  - Remarks
- **Realizing the Analysis Component**
  - Installing BCC
  - Analysis Strategy
  - Implementing Analysis Logic

Implementation – Installing BCC

- With the suitable platform in place, the last thing to install was BCC
- A very convenient way is to do so by using the open source project *Androdeb (adeb)*
  - Creates a debian based chroot environment to install all dependencies to build subsequently build BCC on the platform

Implementation – Analysis Strategy

- Focus of this work is not labeling apps with *uses/doesn’t use packer*, but to showcase if eBPF based tracing can collect sufficient information upon which such labeling could take place
- Further, the analysis method should not have to be tailored towards every application in order to work properly
- So a more general inspection strategy had to be established

Implementation – Analysis Strategy

- The information gathered by the analysis should convey a coarse understanding of the software
  - File interaction
  - Loading code into memory
  - etc
- Unpacked contents should also be retrieved during the analysis

Implementation – Analysis Strategy

- Assumption for this work:
  - Packed malware initially hides harmful code (Dex/ELF) and will at a certain point unpack and load this code into process memory
- User programs (incl. packers) have to heavily rely on system calls, so a viable strategy would be tracing system calls associated with unpacking routines and code loading

Implementation – Analysis Strategy

- In this project the following system calls (a.o.) are traced:
  - `open`: file interaction
  - `unlink`: file removal (packers might delete after use)
  - `mmap`/`mprotect`: associated with code loading
  - `execve`: not necessarily used, but very interesting
- Also library calls are considered:
  - `dlopen`/`android_dlopen_ext`: code loading (ELF/DEX)

Implementation – Analysis Strategy

- Limits of this strategy:
  - Methods of unpacking not using any system calls/library calls
  - For example, code content is written into a executable memory region via a while-loop and direct addressing
    - Polymorphic code
  - However, this affects many analysis strategies
- System call symbols may change between kernel versions
  - Porting between versions not necessarily reliable

Implementation – Analysis Component

- Analysis Component consists of two parts:
  - eBPF logic (eBPF C)
    - Implements the logic carried out within the VM
    - Retrieves parameters from system calls, etc.
  - Management logic (BCC python bindings)
    - Compiles eBPF logic and loads the bytecode into the VM
    - Retrieves output from VM and derives additional information

Implementation – eBPF Logic

- For each tracing target a separate eBPF C script is created
- All relevant scripts will be combined at analysis start
- Retrieve system call parameters, return values and metadata
- Results are submitted into a eBPF ringbuffer, which is polled by the management logic
  - While highly performant, the ringbuffer can occasionally “loose” event samples

Implementation – eBPF Logic

- However, In this form, the analysis component would receive events from all processes currently running on the system
- In Android each installed application runs under a unique UID
- This characteristic is used in the eBPF scripts to filter out any irrelevant events not matching a specified UID

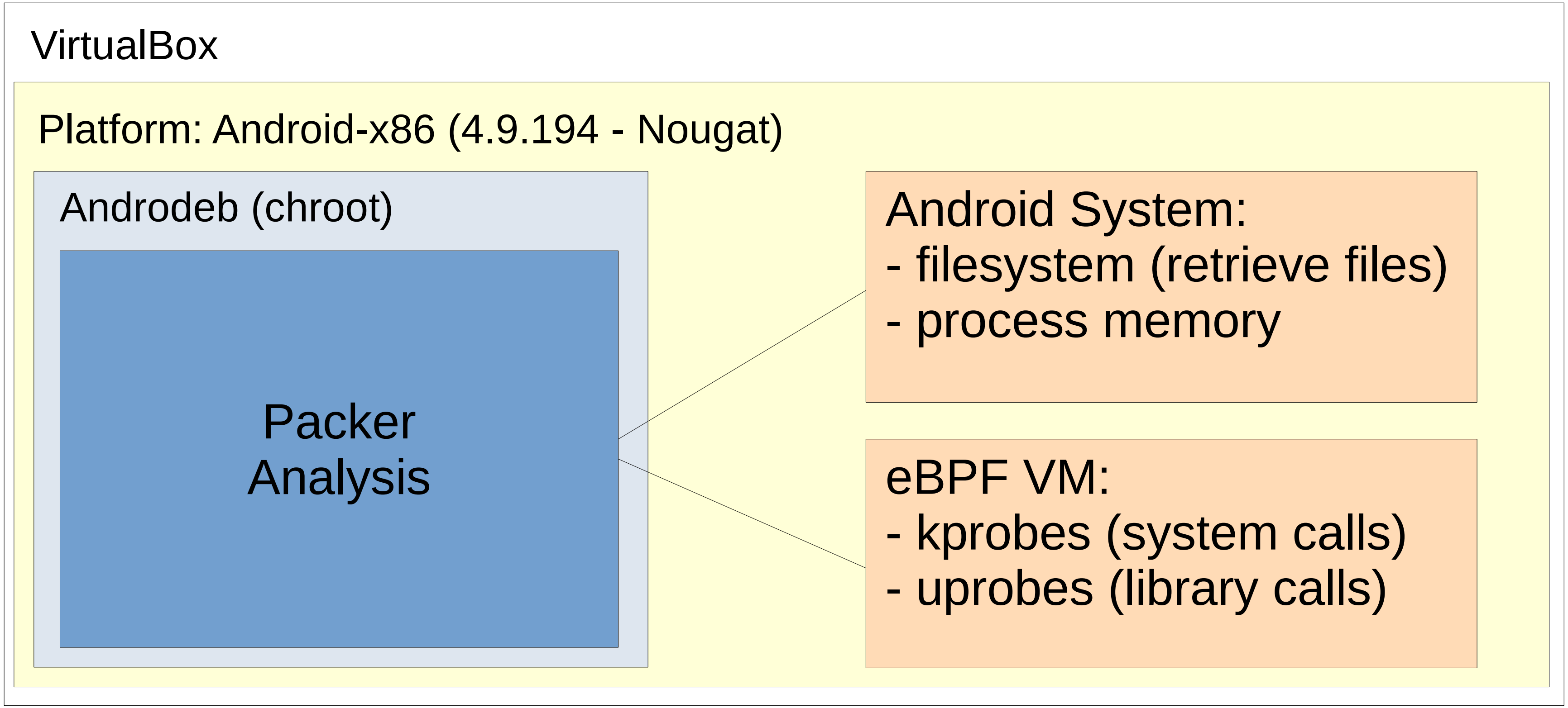
Implementation – Management Logic

- Central component
- Compiles eBPF scripts into bytecode
- Loads bytecode into VM and registers tracing targets
- Retrieves / processes tracing event outputs
- Retrieves unpacked files (from file or from process memory)
- Enriches data (e.g. with filemagic)
- Collects/persists data (Format: JSON)

Implementation – Management Logic

- Problems:
  - Since a direct access to, e.g., files within the BPF VM is not possible, the Management Logic is tasked to retrieve unpacked contents
  - However, this is subject to race conditions , as files might be deleted or memory regions manipulated during the copy attempt
  - This makes the analysis not reliable in that regard

Implementation – Analysis Component



Implementation – Visualization

- Tracing a program can produce enormous amounts of data
- Makes it difficult to understand/interpret the dataset
- In this work, Elasticsearch and Kibana is used have a powerful data exploration and visualization toolset at hand
- If time permits it:
  - Small demo at the end

Implementation – Remarks

- A recurring problem of the Analysis Logic was tracing 32 bit processes
- In 32bit processes different system calls are triggered for compatibility
- However, tracing the 32bit counterpart of system calls often did not yield satisfactory results (parameters were empty)
- To solve this, functions residing “deeper” in kernel had to be traced
- Further exacerbates portability between kernel versions

Evaluation

- Foreword
- Analyzing Samples
- Comparison to strace

Evaluation - Foreword

- Evaluation difficult, as no meaningful metric can be given to estimate the usefulness
- In order to give an impression of eBPF based tracing of packers, the evaluation consists of two parts
- By applying it to samples
  - Showcasing how it can help the task of reverse engineering
- By comparing it to strace
  - Showcasing the abilities/limits

Evaluation – Analyzing Samples

- To give an impression here, samples with a ground truth were analyzed
- That is, apps were the unpacked files have already been retrieved
- For example, a sample app was taken from a Fortinet blog post, which showcased how to retrieve an unpacked file
- This sample was analyzed with the this analysis logic and it was able to retrieve the same file/archive (directly from disk and from the memory region) as the blog post.

Evaluation – Analyzing Samples

- In the Fortinet blog post, the author used Frida to manually hook a certain function to retrieve the file
- In the analysis software this was done automatically (as the respective traces were in place, and race conditions did not interfere with the retrieval)

Evaluation – Comparing to strace

- In this comparison, only the logic running within the VM is considered
- strace is a popular debugging tool, that uses the ptrace system call to trace (among others) the system calls of a specific process
- A downside of strace is, that it is detectable by the inspected program (as only one ptrace can be active at a time, the process could try to trace itself)
- eBPF is less detectable here, as the filtering happens in the kernel VM and not in the process itself

Evaluation – Comparing to strace

- Further, eBPF is able to trace library calls, which strace cannot do
  - This gives eBPF the potential to be an even more powerful debugging tool
- In a direct comparison, the biggest disadvantage of using eBPF is, that there is no actively developed project, which can be used as a basis to expand upon
- Therefore, (300+) individual system call traces need to be implemented manually before being able to compete with strace

Evaluation – Comparing to strace

- This and the problems stated previously (e.g. 32bit processes, portability issues) means, that considerable time and effort needs to be invested to create a tool, that roughly produces the same data a strace (if not considering library calls)

Conclusion

Conclusion

- On it's own, eBPF is not able to provide the necessary spectrum of information to be useful in an analysis of packers
- More in depth information has to be gathered from routines running outside the BPF virtual machine
- However, this makes the analysis prone to race conditions and thus unreliable

Conclusion

- When taking away these external components, the system becomes more superficial (and less useful for specialized packer analysis) and more reminiscent of strace
- However, to be comparable, 300+ system calls may need to be implemented, which is very time consuming (considering the problems)
- Even worse, kernel updates may make the implementation obsolete due to changing kernel symbols

Conclusion

- However, eBPF can become very powerful, if library call tracing is factored in
- And despite the mentioned drawbacks, the presented system can still be used to get a coarse first glance at an packed application
  - In addition to “a chance” to retrieve the unpacked contents
- This “quick test” can help an engineer to prepare a more in depth analysis

Conclusion

- Keep in mind, this work only gives an impression on how eBPF fares in context of the used analysis strategy and how it compares to strace
- There may be some other strategies, where the use of eBPF is more beneficial

Conclusion – Future Work

- Implementing more system call
- Using USDT (User statically defined tracing) to get insights into the Java parts of applications
- Testing the system on an actual device
- Experimenting with eBPF’s capability to read/modify user memory:
  - Placing trampolines for more efficient hooks
  - Is it possible to retrieve file contents this way?

If you are interested...

- ... you can find the project including guides and further information under:

https://github.com/JagwarWest/lab-ebpf-based-tracing-for-packer-analysis-public

Thank you for your attention

- Any questions?