TED: A Container Based Tool to Perform Security Risk Assessment for ELF Binaries

Daniele Mucci[1], Bernhards Blumbergs[1,2]
[1]*Centre for Digital Forensics and Cyber Security*
Tallinn University of Technology
[2]*CERT.LV, IMCS UL*
mucci.daniele@gmail.com, name.surname[at]cert.lv

Keywords: ELF binary analysis, GNU/Linux system hardening, Vulnerability assessment, Software containers.

Abstract: Attacks against binaries, including novel hardware based attacks (e.g., Meltdown), are still very common, with hundreds of vulnerabilities discovered every year. This paper presents TED, an auditing tool which acts from the defense perspective and verifies whether proper defenses are in place for the GNU/Linux system and for each ELF binary in it. Unlike other solutions proposed, TED aims to integrate several tools and techniques by the use of software containers; this choice created the necessity to compare and analyze the most popular container platforms to determine the most suitable for this use case. The containerization approach allows to reduce complexity, gain flexibility and extensibility at the cost of a negligible performance loss, while significantly reducing the dependencies needed. Performance and functionality tests, both in lab and real-world environments, showed the feasibility of a container-based approach and the usefulness of TED in several use cases.

1 MOTIVATION FOR THE PROJECT

From the 1st of January 2017, around 150 different vulnerabilities involving stack or heap overflows and format string bugs have been reported (NIST, 2018). In addition, during this period several hardware vulnerabilities which also involved binaries have been found (e.g., Spectre/Meltdown, Throw-/NetHammer). Also reverse engineering should always be considered as a possible option for targeting binaries, especially where intellectual property is involved. Considering the enormous diffusion of GNU/Linux based systems, it is clear that protecting ELF binaries in such environment is of critical importance, and therefore this is the primary purpose of this project.

2 INTRODUCTION

Many tools and techniques exist to protect systems or binaries from a wide range of attacks, and many more are continuously developed. However, despite the large number of protection mechanisms designed, a relatively small and consolidated set of features is commonly used and deployed. This set includes defenses such as Address Space Layout Randomization (ASLR), Data Execution Prevention (DEP, NX/XD), Stack Smashing Protector (SSP) or Stack canaries. For the most recent Linux kernels, this set can be extended with the microcode to protect against some Spectre variants (e.g., Meltdown). On the other hand, verifying whether all these measures are in place is usually done either manually or by the aid of some tools (e.g., *Radare2*, *Lynis*). This action has to be performed separately for each tool used or individually for each binary, making it time consuming, error-prone and inefficient. Moreover, low-level tools often require specific dependencies for libraries or other tools, which might not always be possible to satisfy or whose installation might require additional time and effort.

The solution proposed in this paper consists of using software containers to address the issues mentioned, specifically to integrate and orchestrate multiple tools, to collect and process the different results, and most of all to greatly reduce the dependencies required, allowing to individually pack the needed tools with their dependencies, thus eliminating possible conflicts and the need for their installation. Given the fact that software containers are still rarely used

outside cloud environments, a significant amount of work consisted in analyzing and comparing the various available container engines to select the one that provided the needed functionalities and at the same time did not represent a heavy dependency itself. The tool here presented, TED, implements such solution using Docker containers to perform all the necessary tests aimed to verify whether a selected set of defense measures are in place, and to consequently assess the risk associated with each binary present in the system. Proposed solution would be practically applicable to the cyber security areas such as, vulnerability assessment, incident response, system triaging, and security baseline establishment.

This paper provides the following contributions:

1. discussion of software containers as execution environment and evaluation of the different container engine platforms in the context of vulnerability assessment; and
2. container technology implementation for ELF binary security assessment in an open-source toolset TED.

This paper is organized as follows: Section 3 gives an overview of related work; Section 4 describes the container engine selection process, the design and the implementation of TED; Section 5 provides the result of the evaluation process; Section 6 concludes this paper.

3 RELATED WORK

Extensive research already exists on binary security, both from the attack and from the defense perspective. However, very few projects, with a purpose similar to the one presented in this paper, were found.

BitBlaze is a platform developed by Song et al. (Song et al., 2008) which uses both static and dynamic analysis to extract a wide range of security information from a program, without taking into consideration the defense measures applied and relying on custom techniques. The main purpose of BitBlaze is to detect possible vulnerabilities in the program and to identify their root cause, rather than determining what security measures are applied. BitBlaze has several components, in particular a static analysis tool (VINE, available and not maintained for 4 years) and a dynamic analysis component (TEMU, available and not maintained for 3 years). YoungHyun et al. (Choi et al., 2015) in 2015 developed a project called DBA (Dynamic Binary Analyzer), capable of dynamically detecting vulnerabilities in binaries with taint analysis, which targets x86 (32-bit) Windows binaries. This project focuses on finding vulnerabilities or detecting exploitation at runtime. To perform its analysis, DBA uses QEMU virtual machine to emulate the execution environment for a single binary. TEASER by Ulrich (Ulrich, 2017) is a system, which aims to assess the exploitability of binaries, performing a vulnerability assessment from the perspective of an attacker. TEASER is limited to identifying memory corruption vulnerabilities and is meant to ease the process of detecting bugs which might lead to exploits. It is built on top of other tools, such as *Valgrind*, *PANDA*, *ASan* and *LLVM*, and uses *QEMU* emulation for some steps of its execution. Tang et al. (Feng-Yi et al., 2016) and Wang et al. (Wang et al., 2017) in their projects focused on binary security analysis in terms of performing a diagnosis of memory vulnerabilities. The two projects which can be compared with TED are *checksec.sh* [1] and Lynis[2]. The first is a *Bash* shell script, which shows technical information, including whether some security measures are applied, regarding a binary, a loaded library or the kernel. The main script is not maintained anymore, but a *forked* and maintained version[3] exists. Lynis, on the other hand, is a software aimed to audit, hardening and testing for compliance Unix systems. The software runs a wide range of tests according to what tools are available on the system, and it is publicly available.

In addition to the presented tools, there is a conspicuous number of proposals to protect binaries from a wide range of attacks, however, an evident gap between the academia and the industry emerged. This means that virtually all the novel tools or techniques, such as (Marco-Gisbert and Ripoll, 2013; Solanki et al., 2014; Younan et al., 2006; Chen et al., 2017; Novark and Berger, 2010), developed in the academic environment, independently by their efficacy and security impact, are either unused or extremely rarely deployed in the production environments.

In the related work, multiple limitations and drawbacks have been identified, such as, the need for specific and numerous dependencies, the use of heavy virtualization technologies (e.g., KVM/QEMU) and their configuration, the support only for Windows binaries or for 32-bit architectures, the focus on single binaries rather than on the whole system, the lack of automation and the need of user interaction and finally the use of technologies that make the tool not portable, not easily extensible or not suitable for cloud environments. TED aims to address all the gaps identified by bundling all the tools and dependen-

[1] `http://www.trapkit.de/tools/checksec.html`
[2] `https://cisofy.com/lynis/`
[3] `https://github.com/slimm609/checksec.sh`

cies needed inside Docker containers, which not only make the application extremely portable, but avoid the overhead of an hypervisor, reducing drastically the performance loss; given this approach, the set of dependencies of TED is fixed and it is limited to the Docker platform and a small number of Python packages. Furthermore, TED focuses on 64-bit executables and aims to automate the scanning process, without requiring user interaction, in order to grant scalability to the project and to allow users to run TED periodically. Finally, TED not only collects and integrates information about both the system and the binaries in it, but also provides an assessment based on the information gathered, to allow for easier planning and better focus on critical parts of the system.

4 DESIGN AND IMPLEMENTATION

The design and implementation of the tool proposed here, TED, necessitated of several steps. First, the most suitable container engine for the context in which TED operates needed to be selected, then, it was required to individuate the defense measures and techniques to integrate in the tool, and furthermore to establish a corresponding scoring system to measure the system's ELF binary state. Finally, the tool had to be implemented.

4.1 Container Engine Selection

The idea of software isolation was born during the 1980s, however, the first use of containers dates back to the early 2000s. Despite this, only from 2015 containers have become a popular and ubiquitous technology, therefore the market is still relatively young, leading to the flourishing of many container engines and orchestration systems. Among these, five available platforms have been taken into consideration: Docker[4], Rkt[5], LXC[6], OpenVZ[7] and Garden[8]. OpenVZ and Garden have been discarded before the detailed evaluation for the following technical reasons: OpenVZ has limited functionalities if it is not run on a specific kernel, imposing a heavy constraint for a tool that aims to be generic and portable; Garden on the other hand is not yet a mature project and it is virtually not used, despite it aims to represent an alternative to Docker.

The remaining platforms – LXC, Docker and Rkt, have been evaluated using three parameters:

1. Availability: what needs to be done to get and install the platform, what dependencies need to be satisfied and also whether there are public repositories from which container images can be obtained.
2. Functionality: whether and how the container engine provides capabilities such as building images, collecting logs and events and sharing portion of the file system with the host.
3. Performance: the overhead caused by executing code inside a container for each engine is measured running 1000 times a test script and comparing its execution time with a baseline established running the same script on the native machine.

The most important criterion is the functionality, since a lack of features would impact the capabilities of TED or increase its complexity. Following the functionality, the availability is considered, as this impacts the dependencies needed by TED, which are meant to be as few as possible. Finally, performances are considered of secondary importance, since the difference in performances between technologies which are fundamentally similar is expected to be small.

Docker. Docker is the most popular container engine and the current industry standard. It can be installed in most cases from the package manager and does not need any particular Kernel feature, although *root* access is necessary to run the Docker daemon. The necessity of root access and its security implications are well documented[9] and require special attention. It is important to note that Docker is included in many cloud-oriented operating systems, such as CoreOS Container Linux [10] and RancherOS [11]. Container images that can be run by Docker are publicly available online from a public repository called Docker Hub; the Docker Hub makes extremely easy to find, run and publish an extensive set of images through HTTP. The Docker engine is focused on providing an execution environment for one process only, meaning that in general there should be a 1:1 ratio between Docker containers and applications running, and also that a container terminates when the process with PID 1 exits. Docker offers a rich set of functionalities, including an API to build, run and manage containers; in particular, it offers a simple way to

[4] https://www.docker.com/
[5] https://coreos.com/rkt/
[6] https://linuxcontainers.org/
[7] https://openvz.org/
[8] https://github.com/cloudfoundry/garden

[9] https://docs.docker.com/engine/security/security/#docker-daemon-attack-surface
[10] https://coreos.com/os/docs/latest/
[11] https://rancher.com/rancher-os/

start and stop containers, extract logs or outputs from them, build new container images and share portions of the file system with the host.

LXC. LXC is defined as "a userspace interface for the Linux kernel containment features" (LXC-Doc, 2018). Unlike Docker, the purpose of LXC is to provide a full virtual environment without the burden of running a separate kernel and a hypervisor. Thus, in this case, several applications can run inside one single container. There are few hard dependencies for LXC which are likely to be fulfilled on most systems and therefore do not impose a hard constraint. Installing LXC is generally possible through the package manager, although no operating system comes with LXC pre-installed. It is possible to import public "templates" for different operating systems, but there is not any public repository for already made containers, however, it is still possible to distribute images via HTTP with a private web server. The functionalities offered by LXC are many and powerful, but in general require a careful configuration. The building process for an image is very different from the one offered by Docker: with LXC it is possible to "export" and then "import" an image. The export produces usually a *tar.xz* file or two *tarballs* that can be imported, which are more similar to a snapshot of the running container, rather than to a fresh image. Information about container events can usually be found in the host journal, while, unless manually specified, container logs need to be extracted directly from inside the containers. It is possible to share a portion of the file system with the host by simply mounting it inside the container, provided that a correct permission configuration is in place.

Rkt. Rkt (Rkt-Doc, 2018), read "rocket", is a container engine developer by CoreOS Container Linux team, and – apart from low-level details – it is very similar to Docker. Rkt containers are meant to run a single application on which their lifecycle depends. Rkt is available from the package manager in most Linux distributions, with the exception of Ubuntu and CentOS, and it is pre-installed on CoreOS Container Linux. The only dependency for running Rkt containers is a kernel version 3.18 or later. Rkt can run both Application Container Images (ACI) and Docker containers. For the firsts, the location of the container needs to be known *a priori* to be downloaded, since a public repository of images is not available, while for the latter it is possible to use the Docker Hub. The functionalities of Rkt, especially from the perspective of this project, are very similar to the ones offered by Docker, except for building a new image. Since Rkt can run Docker images, it is possible to use Docker build feature, but in a more general process, the build of a new container needs to be done through *acbuild*[12] tool, that appears to be not maintained. The logs of Rkt containers can generally be read through the host journal and most of the operations necessary to manage containers can be performed through the Rkt Command Line Interface (CLI) while an API with read-only capabilities is available. Events and log collection is done in a similar fashion to Docker, and similarly volumes are used to share the file system with the host.

Performance overview. To measure and compare the performances of the different platforms, a simple script which emulates some functionalities of TED has been created. This script has then been executed 1000 times on the native machine and inside a container run with each of the engines presented. The average execution time on the native operating system was 3.8s, whereas Rkt and Docker offered faster performances – 3.33s (-12.4%) and 3.5s (-7.9%) respectively. The slowest result was obtained by the LXC container, which took an average of 3.9s (+2.6%) to execute the script.

Conclusions. Taking into account both the criteria to evaluate the platforms, the information gathered and the results from the performance tests, it is possible to observe that LXC – in addition to being the slowest platform – requires a heavier configuration from the host side compared to Rkt and Docker. In particular, the work required to configure the container network is a consistent overhead to the deployment of the application. Moreover, the way LXC templates are distributed implies that not only a public service needs to be set in place, but also that a new template should be downloaded every time an update to the application is performed. Rkt and Docker, on the other hand, offer similar functionalities and performances, making the choice between them less obvious. However not only the Rkt CLI offers limited capabilities compared to the Docker API, but Rkt has also a much less capillary diffusion and user-base than Docker, it is less tested and it is used mainly in the context of CoreOS Container Linux, whereas Docker is widespread and battle-tested on a vast array of platforms. In the context of this project, Docker does not present any of the drawbacks identified in the other platforms, offers all the functionalities needed, including good performances and a convenient way to manage and distribute images. For all the reasons above, it is clear that Docker results the most suitable candidate to be used in TED.

[12] `https://github.com/containers/build`

4.2 Defense Measures Selection and Scoring System

The main functionality of TED is to verify whether certain protection mechanisms for binaries are in place, therefore selecting the appropriate measures becomes of critical importance to ensure the quality of the results. To choose the most suitable defenses, both their popularity and their efficacy have been taken into consideration, to avoid as much as possible false positive results. It is possible to divide the defense measures taken in consideration for being integrated in TED in three main categories: system defenses, ELF defenses against vulnerability exploit and ELF defenses against reverse engineering.

System wide defenses. This category includes those measures used by the kernel or applied to all the executables running in a given machine. Their purpose is to offer some level of protection to the environment in which the binaries run, without considering the defenses that each specific binary might or might not include. These include Address Space Layout Randomization (ASLR) and its analogue for the kernel (KASLR), hardware based memory protection (e.g., NX – No eXec, XD – Execute Disable) and a novel mechanism originally designed to increase kernel security, which later was recognized also as a protection from the Meltdown attack, KAISER/KPTI (Gruss et al., 2017). (K)ALSR and NX/XD have been chosen because they offer a good degree of protection from a wide range of memory exploitation attacks, including those belonging to the *ret2** family; however, more advanced attacks exists (e.g., ROP and derivatives) but the defense measures against this kind of attacks are not commonly found on systems and therefore have not been taken into consideration. Finally, KAISER/KPTI at the moment is the only existing defense against Meltdown for Linux systems, and given the critical impact of such attack, it has been included in TED.

Executable defenses against vulnerability exploits. This set includes the measures which are applied individually to each executable, for example during or after compilation time, and that aim to protect them from exploits that target vulnerabilities in the code. These measures offer a certain degree of protection independently from the system in which the binary is run. The most popular and commonly used techniques are software canaries, in particular the Stack Smashing Protector (SSP), the software based Data Execution Prevention and its corollary, the W^X (Writable XOR eXecutable) principle. Similarly to what it has been discussed previously, more advanced and novel protections for binaries have been designed, but none of this can be considered as a standard or as commonly deployed, and therefore they have not been integrated into TED.

Executable defenses against reverse engineering. The last category contains the protections aimed to counter or make more difficult the reverse engineering of a binary. The only measure which TED checks is the stripping of the binary. Stripping is an extremely weak defense, which makes the debugging of the program slightly more difficult, but it is the only common measure applied. The reason for which no other techniques are considered is that no standard binary protector for Unix systems exist, and that the focus of the research in this field is on software obfuscation. Unfortunately, TED is run on binaries whose content is in general unknown, and therefore it is not possible (or very complex) to verify whether the code is obfuscated or not. This limitation effectively leave stripping as the only measure verifiable by TED.

Additional tests. In addition to verify whether the measures mentioned are in place, TED also checks if the kernel is vulnerable to known exploits or to different Spectre variants. The rationale behind this choice is that this is not only a way to verify kernel binaries security, but also a decision which derives from the observation that generic binary attacks can have higher impact on machines where the kernel is vulnerable to exploits, especially to Privilege Escalation exploits. Finally, Spectre attacks could be used to compromise the machine leveraging hardware faults that are not mitigated by the defenses mentioned above (except for Meltdown).

4.3 Scoring System

To convert the information that TED gathers into a value that helps the TED user to make plans, two scoring systems are defined – a system score and an ELF score. Each score is computed on a total of 100 points, where a higher score implies a higher chance of exploitation, while 0 means that all the defenses that TED is able to verify are in place. The assignment of a score to each (lack of) defense is purely qualitative and decided after a careful consideration. The system score is so composed:

1. Kernel exploits (40 points). For each confirmed kernel exploit found, a score of $15/2^{n-1}$ is assigned, with $n \in [1,\infty]$, which represents the *n-th* exploit found. Each potential kernel exploit found will add $5/2^{n-1}$ points. Kernel exploits usually have a very big impact on the security of the machine. However, establishing a linear dependency between the number of exploits and the score would generate misleading results. For this

reason, the first exploit found will significantly increase the risk score, while each additional exploit will contribute with a continuously lower weight.

2. Spectre (20 points). Each variant of Spectre to which the system is vulnerable adds 3 points, while 8 points are added if the system is vulnerable to Meltdown. At the time of this writing, 5 variants of Spectre are known. Although Spectre attacks can lead to a severe compromise of the system, because there are not yet known exploits for them, the score assigned is relatively low. Meltdown contributes with a higher score because it is relatively simpler to exploit.
3. ASLR (20 points).The ASLR configuration can lead to a maximum of 20 points, depending on how the randomization is enabled. Specifically, 20 points are assigned if ASLR is disabled, 15 if it is partially enabled (i.e., only on stack, virtual dynamic shared object page, and shared memory), 5 if it is enabled also for the heap and data segment and 0 points if ASLR is enabled fully and additional patches have been applied. ASLR is an effective measure that should be enabled on all the systems, however, the different ASLR configurations restrict or enlarge the attack surface, and therefore are taken in consideration in the assignment of the risk score.
4. NX/XD (20 points). The lack of support from the CPU for the NX/XD bit on memory pages adds 20 points, while no points are added if its support is confirmed. The capability to restrict the execution of certain memory page is a very important measure to stop the simplest attacks and to build additional defenses, and therefore a significant score is assigned in case of missing support for NX/XD.

The binary score is composed as follows:

1. DEP/WˆX (50 points). If the binary is compiled with the stack marked as executable (violating DEP), 30 points are added. If there is any section both writable and executable (violating WˆX), 20 points are added. The possibility to write on the stack contributes with a big score as it enables the most basic attacks. The capability to write on an executable memory section is still a severe vulnerability but in general it is less trivial to exploit and therefore contributes with a lower score.
2. Canaries/SSP (40 points). If the binary is compiled without canaries and without using the Stack Smashing Protector, 40 points are added. Canaries are an orthogonal measure to DEP and also represent an important measure to detect exploit attempts. For this, the lack of canaries increases significantly the risk score of the binary.
3. Stripping (10 points). If the binary is not stripped, 10 points are added. The low score reflects the low effectiveness of stripping a binary to counter reverse engineering.

4.4 Implementation

The implementation of TED has been performed using mainly *Python* language, for which a library [13] which implements all the operations normally performed through the *Docker* command or API is available. It is important to note that TED is especially suitable for cloud environments because of several reasons: the first is that in such environments Docker is commonly present, allowing TED to be deployed in a matter of seconds; the second reason is that even though the tools run inside containers, TED can scan and analyze all the binaries in the system, including those inside other containers, without the need to modify the employed container images or their configuration. Moreover, the approach of TED to analyze both the system and the binaries is especially relevant in containerized environments since most of the container-escape strategies rely on kernel/system vulnerabilities. The program is composed by several modules that perform specialized operations. These operations include the acquisition of user input, the collection of the binaries to analyze, the execution of TED's checks, the collection of the results and the score computation. Each check that TED performs aims to verify whether one or more defense measures are in place, therefore they match the defense techniques mentioned in the previous section. In total, five different tests are implemented at the moment, each of which is run inside its own Docker container.

Kernelpop test. A custom version of Kernelpop[14] is used to verify whether the kernel is vulnerable to known exploits. This tool has been chosen because it provides fast and accurate results, it is maintained and its code is publicly available.

Spectre test. The bash script *spectre-meltdown-checker.sh*[15] is used to verify whether the system is vulnerable to 5 different Spectre variants. This script is comprehensive and actively maintained, and provides results in *json* format and therefore it is very suitable for being integrated inside TED.

NX/XD support test. The standard */proc/cpuinfo* is used to verify whether the CPU supports the *nx* bit. Given the fact that containers share the kernel with the

[13] https://github.com/docker/docker-py

[14] https://github.com/Sudneo/kernelpop

[15] https://github.com/speed47/spectre-meltdown-checker

host, the information obtained by */proc* virtual files is accurate and no external tools are needed.

ASLR test. In this test both the standard *sysctl* tool and a custom binary are used to detect the ASLR configuration active in the system. While the *sysctl* result provides the information from the kernel perspective, the custom binary uses a direct approach to observe the pattern in addresses assigned to the stack, to *malloc()* calls (heap) and to the environment variables. Analyzing these address and checking for repetitions allows to detect if ASLR is disabled or if it is enabled partially. The two perspectives are compared to provide a more accurate result for one of the most critical defenses.

ELF test. In this test *rabin2*, one tool which is part of the *radare2*[16] framework, is used to extract some information from the binaries, together with the standard *objdump* tool. The purpose of this test is to verify whether the binary has been compiled with DEP and W^X, whether canaries are used (and in particular the SSP) and if the binary is stripped. The first test consists in inspecting the flags of the binary sections/segments, the second consists in disassembling the binary and verifying the presence of canary check functions, while the binary is considered stripped if no symbol sections are found. *Radare2* has been chosen because it is one of the most popular tools for reverse engineering in Unix environment and it is actively maintained.

The main accomplishment of TED is not only to integrate and orchestrate the tools mentioned, but also to use containers to pack the needed tools with their dependencies in separate environments. This strategy allows to run easily these tools and also to use at the same time libraries or programs with different versions without creating conflicts. To provide such functionality, several Docker images have been built, while in the cases where a specialized image was not necessary the public *debian:latest* image has been used. The code of TED is publicly available and can be found on a public Github repository[17] and all the Docker images are available either on the public Docker Hub [18] or on the authors' Docker Hub page[19].

5 EVALUATION

To verify the feasibility of a container based approach for a security tool and the usefulness of the tool presented in this paper, several tests with different scopes and use-cases have been performed.

5.1 Performance Benchmark

The first test is a quantitative analysis with the aim of computing the overhead caused by running the needed tools inside containers. The four heaviest checks (excluding the NX/XD support check) that TED performs have been executed both inside a container and on the native operating system, measuring and comparing the respective execution times. Each check has been executed 100 times, and the average value has then been computed. The results are summarized in Table 1, where the average value for each different check is reported. It can be observed that the overhead added by running inside a Docker container is noticeable in relative terms; however, for each of these tests, a new container has been created and then destroyed. It was therefore possible to suppose that the overhead was caused by the Docker bootstrap routine. To confirm this hypothesis, the two checks with the major absolute difference between values have been modified to be executed inside the same container and the test has been repeated. The result obtained, also shown in Table 1, clearly confirms the hypothesis formulated. If the container bootstrap process is not factored in the execution time, the difference between running the checks inside containers or on the native machine becomes negligible. Moreover, the assumption of reusing a container for multiple tests does not represent a constraint, in fact, TED uses the same approach for its own execution. In conclusion, the performance analysis showed that executing the needed tools inside a container the way that TED does, causes an overhead in the order of few seconds per tool run, which in a non real-time processing context and compared to a global execution time potentially in the order of hours, can be considered negligible; therefore, the container based approach can be considered feasible from the performance perspective.

5.2 Functionality Tests

The second phase of testing included a set of real-world experiments to verify the usefulness of TED in real environments. For this, two main scenarios have been used: the first consisted of using TED to assess the security of the binaries present on a public web server and to consequently establish an action plan to resolve eventual issues found; in the second case, TED has been used on a portion of the infrastructure used for the NATO Locked Shields 2018 cyber exer-

[16] https://github.com/radare/radare2
[17] https://github.com/Sudneo/TED
[18] https://hub.docker.com/
[19] https://hub.docker.com/u/sudneo/

Table 1: *Summary result for the benchmark checks.*

Check	Physical	Docker	Relative Δ	Absolute Δ	Docker no startup	Relative Δ	Absolute Δ
ASLR	0.031s	0.469s	+1418%	0.438s	-	-	-
Kernelpop	0.112s	1.559s	+1284%	1.447s	0.134s	+16.4%	0.022s
Spectre	4.745s	6.066s	+27.8%	1.320s	4.485s	-5.8%	0.260s
ELF	0.00060s	0.00055s	-8.4%	0.00005s	-	-	-

cise [20] to determine whether it would be a good fit in a complex environment.

Public web server. TED full scan on the web server took 84m and 2s to complete and found the issues summarized in Table 2, where the actions that have been established to mitigate them are also reported. In Table 2 it is possible to observe that all the actions defined involve the system and the kernel, while no actions were planned for the ELFs compiled without defenses. The reason for this is multifaced; first, most of the binaries were used for local commands and were not facing the public. Second, none of the binaries had the *setuid* bit set, and finally, to take action and recompile the binaries, a more in depth inspection was necessary to make sure that recompiling the code with DEP or SSP enabled would not have broken any compatibility. For a matter of simplicity, further actions like configuring SELinux profiles or AppArmor for specific binaries, had not been considered. After performing the actions presented in Table 2, all the addressed issues have been resolved, and the following TED scan, which took 152m and 52s, reported a much lower risk level. It was therefore possible to conclude that TED allowed to effectively detect vulnerabilities in the system and also it facilitated the creation of an action plan composed of informed choices.

Locked Shields infrastructure. This test consisted of running TED on two sets of machines used during the cyber exercise: the first set included undefended machines, whereas the other set was composed by two machines which belonged to and had been defended by the winning team. The machines used for testing were web servers which run their applications inside Docker containers. It is worth mentioning that all the four servers used for this test did not have any Internet connection (after the game finished) and were accessible only through SSH once connected to the closed virtualized environment hosting the game-network. This condition presented the opportunity to successfully prove that an Internet connection does not represent a hard dependency for TED. Comparing the results of TED's scans on the first server of each set it was possible to observe that the blue team performed some updates (e.g., *sshd* and *apache2* binaries had been modified) and of course installed new software (e.g., *ossec*, *splunk*), but no major modifications to binaries to reinforce their defenses had been performed. One exception was represented by the *mysql* binary that scored 40 points in the undefended machine, but only 20 in the defended one, where canaries had been added. The most interesting observation is that in both cases the system score reported by TED was the same, in particular, the same kernel vulnerabilities were present in all the machines, among which the most interesting was the *dirtyCOW*. This exploit grants a privilege escalation and is especially relevant in contexts where applications run inside Docker containers, since it can allow attackers to break out of such containers [24]. The same observations could be done for the second pair of machines, where the exact same differences could be observed.

Conclusions. The results of the real-world tests showed that TED has very limited dependencies, can run without an Internet connection, effectively detects vulnerabilities in the system and helps creating an action plan to resolve them. Despite this, it might be needed to integrate more tools into TED to provide more detailed information, for example about what changed in each binary or whether each executable is malicious or not. It is reasonable to conclude that TED could fit in contexts such as The Locked Shields especially if the defending team knows on which binaries to focus (e.g., web server binaries), reducing the scope of the report and improving the capability to quickly assess the risk on meaningful binaries, to individuate possible system vulnerabilities and to correlate the findings.

6 CONCLUSIONS AND FUTURE WORK

In this paper a novel approach to vulnerability assessment – using software containers, is discussed, together with the selection process for the most suitable

[20] https://ccdcoe.org/largest-international-live-fire-cyber-defence-exercise-world-be-launched-next-week.html

[24] https://github.com/scumjr/dirtycow-vdso

Table 2: *TED's findings on the public web server.*

Type	Description	Action
System	Kernel vulnerable to dirtyCOW and dirtyCOW poke variant	Upgrade kernel
System	Kernel vulnerable to CVE20177308[21]	Disable user namespace usage to unprivileged users.
System	Kernel vulnerable to CVE20162384[22]	Update kernel or restrict USB access.
System	Kernel vulnerable to CVE20176074[23]	Disable kernel DCCP module.
System	Machine vulnerable to Spectre variant 1,2 and Meltdown	Upgrade kernel, enable KPTI
ELFs	71 binaries compiled without canaries of which 26 without NX	-

containers platform, and the design, implementation and verification in multiple use cases of a tool, TED, which implements this approach. The planned improvements to TED include automated detection of malicious binaries through the integration of an antivirus such as ClamAV. Moreover, to improve the accuracy of the ELFs tests, the capability to analyze each binary to verify that no unsafe functions are used will be added. From the functionality perspective, despite TED runs all the checks it performs inside containers, the main application still runs on the native machine. Fully containerizing TED, including the supporting functionalities, will reduce the dependencies needed to the sole Docker platform, making the tool even more suitable for cloud environments, where containers are already used in many cases and where the Docker platform often is pre-installed, and would allow TED do be easily deployed with orchestration tools such as Kubernetes, increasing drastically the scope and the target audience. Finally, future work includes research on protections of binaries from reverse engineering, which at the moment are noticeably scarce, exception made for code obfuscation techniques.

REFERENCES

Chen, X., Xue, R., and Wu, C. (2017). Timely address space rerandomization for resisting code reuse attacks. *Concurrency and Computation*, 29(16).

Choi, Y.-H., Park, M.-W., Eom, J.-H., and Chung, T.-M. (2015). Dynamic binary analyzer for scanning vulnerabilities with taint analysis. *Multimedia Tools and Applications*, 74(7):2301–2320.

Feng-Yi, T., Chao, F., and Chao-Jing, T. (2016). Memory vulnerability diagnosis for binary program. *ITM Web of Conferences*, 7.

Gruss, D., Lipp, M., Schwarz, M., Fellner, R., Maurice, C., and Mangard, S. (2017). Kaslr is dead: Long live kaslr. volume 10379, pages 161–176. Springer Verlag.

LXC-Doc (2018). Lxc official reference. `https://linuxcontainers.org/lxc/introduction/`. Accessed on 20/10/2018.

Marco-Gisbert, H. and Ripoll, I. (2013). Preventing brute force attacks against stack canary protection on networking servers. In *2013 IEEE 12th International Symposium on Network Computing and Applications*, pages 243–250.

NIST (2018). Nist nvd, vulnerabilities statistics. `https://nvd.nist.gov/vuln/`. Accessed: 10/06/2018.

Novark, G. and Berger, E. D. (2010). Dieharder: Securing the heap. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, pages 573–584, New York, NY, USA. ACM.

Rkt-Doc (2018). Rkt official reference. `https://coreos.com/rkt/docs/latest/`. Accessed on 20/10/2018.

Solanki, J., Shah, A., and Lal Das, M. (2014). Secure patrol: Patrolling against buffer overflow exploits. *Information Security Journal: A Global Perspective*, pages 1–11.

Song, D., Brumley, D., Yin, H., Caballero, J., Jager, I., Kang, M. G., Liang, Z., Newsome, J., Poosankam, P., and Saxena, P. (2008). Bitblaze: A new approach to computer security via binary analysis. In Sekar, R. and Pujari, A. K., editors, *Information Systems Security*, pages 1–25, Berlin, Heidelberg. Springer Berlin Heidelberg.

Ulrich, F. (2017). *Exploitability Assessment with TEASER*. Msc thesis, Northeastern University.

Wang, R., Liu, P., Zhao, L., Cheng, Y., and Wang, L. (2017). deexploit: Identifying misuses of input data to diagnose memory-corruption exploits at the binary level. *The Journal of Systems & Software*, 124:153–168.

Younan, Y., Pozza, D., Piessens, F., and Joosen, W. (2006). Extended protection against stack smashing attacks without performance loss. In *2006 22nd Annual Computer Security Applications Conference (ACSAC'06)*, pages 429–438.