2017 IEEE 14th International Conference on Mobile Ad Hoc and Sensor Systems

Cybergrenade: Automated Exploitation of Local Network Machines via Single Board Computers

Anurag Akkiraju, David Gabay, Halim Burak Yesilyurt, Hidayet Aksu, Selcuk Uluagac
Cyber Physical Systems Security Lab
ECE Department, Florida International University, Miami, Florida 33174
anurag.akkiraju@gmail.com, {dgabay, hyesi001, haksu, suluagac}@fiu.edu

***Abstract*—In this paper, we introduce a defensive cybersecurity framework called Cybergrenade automating various penetration testing tools to sequentially exploit machines connected to a single local network, all underneath a single application running on a Single-Board Computer (SBC). This takes advantage of the SBC's unique capabilities in a way that manual exploitation simply cannot match. Currently, while many SBCs are being used in research as exploitation tool-kits, the current state of automation of the processes associated with exploitation leaves much to be desired. While this paper describes the Cybergrenade Framework, it can be used as a guideline for future research automating the exploitation process. Cybergrenade allows tools such as Nmap, OpenVAS, and Metasploit tools to be automatically utilized under one framework. Our experimental evolution revealed that Cybergrenade can perform the automation of various pentesting tools under a single application with ease.**

***Keywords*—Automated Penetration Testing, Local Machines, OpenVAS, Metasploit, Nmap, Exploitation, Single Board Computers**

I. Introduction

In the field of Cyber-Security, there is a constantly growing number of Common Vulnerabilities and Exposures (CVEs). Software applications often include numerous identified and zero day vulnerabilities, thus making the machines susceptible to intrusion. As the number of CVEs continues to grow so does the importance of businesses' properly penetration testing their networks, to prevent serious attacks on the networks' machines from occurring. In this project, we introduce the Cybergrenade framework which imitates an attack on a local network for defensive purposes, exploiting the machines using their respective CVEs. To symbolize a real attack on the networks machines, we use a small Single Board Computer which allows the attack to be a covert operation. The defining factor is the automation of various aspects of the penetration testing process all under a single script.

A 'Single Board Computer (SBC)' describes a single circuit board that has a 'complete' ecosystem of computing components such as a microprocessor, Input/Output (I/O), and memory. As computing power has been compacted more and more in smaller and smaller processors, the mainstream availability of SBCs has risen dramatically. SBCs are ubiquitous today and include the various Raspberry Pis, BeagleBones, and ODROIDs. The potential to use these various SBCs, often small enough to fit in one's hand, in many covert activities is great. SBCs can be hidden in many small areas in order to escape visual detection, which makes them perfect to use as exploitation machines. The problem therein lies that if one

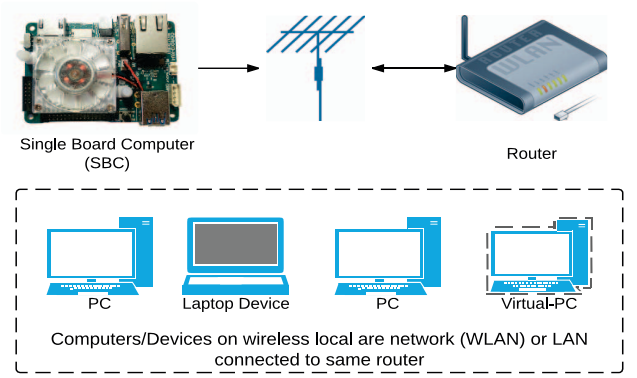

Fig. 1. Cybergrenade System Flow

of the benefits of using an SBC instead of a objectively more powerful full sized computer lies in its potential to be covert, then proceeding to manually find vulnerabilities and fire exploits at other machines in the local network through the SBC would defeat the whole intention of remaining inconspicuous. Therefore, to truly play to the real strengths of SBCs, the whole exploitation process must be as autonomous and self sufficient as possible, only necessitating human interaction in order to connect to a local network in the first place. After this, the SBC would preferably only be interacted with when it's time to take it away surreptitiously from its hiding spot. The overall interaction the SBC has with the local network is shown in Figure 1.

In this paper, we show that this solution has many useful applications, namely being able to be used by many not necessarily familiar with any of the 'made-for-manual-usage' penetration testing tools needed to perform reconnaissance and exploit machines on a local area network. As stated previously, the automation provided as the solution holds true to the core of using SBCs as penetration testing tool kits, as well as providing more flexibility in terms of allowing more covert exploitation of various machines. This project is also a blueprint for automation within various tools while keeping their running logic intact.

When we define the exploitation of local machines, we are, in this case, referring to gaining shell access on the machines without sanction. Our results differ from related works constructing penetration testing devices using SBCs as the exploitation process is automated. In order to autonomously target and 'exploit' machines on a local network, there is integration with a network scanner to find needed information

2155-6814/17 $31.00 © 2017 IEEE
DOI 10.1109/MASS.2017.95

about potentially exploitable machines on the local network. Here exploitation works as a means of empirically verifying the vulnerability. In addition to this, there is integration with a vulnerability assessment scanner in order to find potential weaknesses in the machines that will be tested. Finally, there is integration with a penetration testing framework in order to search for and fire exploits at the identified local machines based on the information given by the network scanner and vulnerability assessor.

The remainder of this paper is structured as follows: In Section II, related works are presented. In section III, we introduce the design of Cybergrenade involving the tools used and how to create the autonomous penetration testing device. Our experiments and the results are presented in Section IV. Lastly, we conclude by presenting a summary of our work, its applications, and its uniqueness in Section V.

II. Related Work

A great deal of researchers have worked on using SBCs as potential pentesting tool kits. However, there are very few researchers who have expounded on a way to automate the pentesting tools within a single SBC to take full advantage of the SBC's capabilities. The researchers in [7] succeed in the automation of OpenVAS in the same way we do using the same XML connection to OpenVAS Management Protocol (OMP) [12] that we elaborate in the next section on how we use it in our framework. Though, in [7]'s case a manual interference is still required, for example, when reading and working off information automatically delivered to a web server, or while picking and choosing targets to scan for vulnerabilities. Our solution for automated OpenVAS/OMP is different and unique in the way we expound on catching the randomized Unique Alphanumeric Identifier, or UID, values sent back by OMP to the SBC as elaborated in Section III of this paper. Furthermore, [7] stops at the point of importing the automated vulnerability scan into MSFconsole and not automating MSFconsole in anyway to do anything with the given information. This paper does describe a way to automate MSFconsole and succeeds in getting MSFconsole to run any command a human would manually run in the same situations. In the case of paper [4], the authors use a method called SSH callback to connect a penetration testing device to a computer outside of the local network. Once connected, they control the penetration testing device and view results on a web application, they designed specifically to display the exploits and vulnerabilities found in their experiments. Like [7], the design in [4] only automates the initial startup processes, this includes scanning the local network and populating the Metasploit database with the discovered hosts, ports, and vulnerabilities. Nonetheless, beyond this initial script, the user must manually try to exploit the vulnerabilities on these machines.

Another work [6] automates MSFconsole successfully in a similar way we end up automating it, by manipulating resource scripts. However, the caveat is that for their paper, it would only work in the case that they knew ahead of time which exploit to run, thereby not being as flexible in terms of our solution in which the resource script file sent with MSFconsole is truly procedurally generated each time we run the script. [6] also does not focus on using these tools in tandem with SBCs, which we are focusing on with this paper. Both [7] and [6] use Python as the chosen scripting language for their works, similar to the Cybergrenade framework.

Besides aforementioned related works, there are two related works focused on security assessment. [2] is one security assessment study that covers next generation wireless mobile networks. Another penetration testing and security assessment work is [3], which utilizes Nmap and OpenVAS software to perform penetration testing like Cybergrenade. In [1]'s technical report, there is a penetration testing system architecture that is similar to the architecture of Cybergrenade.

In addition to all these works, [5] is an instrumental reference for ideas on why to run Kali Linux on any SBC over other Operating Systems. It also explains the manual 'exploitation process' consisting of a network scan, vulnerability assessment, and exploit delivery through SBCs, which is roughly followed in the Cybergrenade Framework, though in an automated fashion.

III. System Design and Implementation

In the Cybergrenade framework, Kali Linux [8], created by Offensive Security, an image of Linux with many pentesting tools already built in, was used as the OS for the SBC-based pentesting device. This OS can be used on every SBC that can handle more than 8 GB of memory, which makes a good fit to base the system on. As most SBCs are powerful enough to run all the tools needed to perform an automated exploitation, some SBCs would simply be faster due to larger amounts of RAM. For example, a Raspberry Pi 3B [14] with Kali Linux on it, hence can run all the necessary tools on it, albeit slowly. In Cybergrenade, we used the ODROID XU4 [11] due to the XU4's 2 GB of RAM, among other things, compared to the Pi's 1 GB of RAM, which helped speed up the runtime greatly. Nevertheless, the Cybergrenade framework can be replicated on any SBC that can host the full version of Kali Linux on it. On all the SBCs, we used a 32 GB Micro-SD card for memory storage, and encountered no problems with running out of memory throughout this project. Furthermore, for its readibility and its native support by many tools, we chose a lightweight scripting language, Python, to connect all the various tools needed together under one executable script. Python also provided tight and easy access to the command shell in terms of sending commands to be executed and was easily able to spawn various threads as needed.

A. Tools Used

To scan the network and return information about the IP Addresses, Ports, Protocols, and OS used for the local machines, Cybergrenade uses 'Nmap', a free and open source network mapper natively supported on the Kali Linux image. For a vulnerability assessment scanner, the options natively supported on Kali Linux were not very flexible as they mainly focused on web-based vulnerability assessment, which we wanted to stay away from in case the local network is a closed system in of itself and the machines on the local network could not connect to the Internet. We also aimed to make sure that the vulnerability scanner we used was free so that it would be available for anyone to download using the proposed solution; as well as open source. For these reasons, we settled on 'OpenVas', a free service provided by 'Greenbone Networks'.

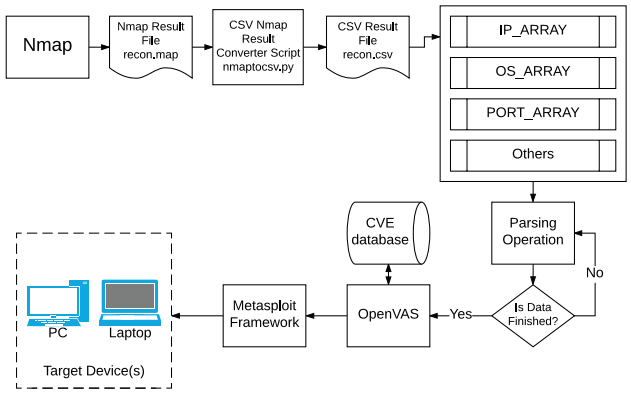

Fig. 2. Architecture of Cybergrenade

As Kali Linux came with the 'Metasploit Penetration Testing Framework' built in, it was chosen to find and deliver exploits to the local machines using information given to it by the other two tools. All of the penetration-testing tools used were made generally to be manually used, with little, if any, support for automation, which presented a serious problem and demanded some unique solutions as we progressed.

B. System Workflow

As stated earlier, the script that is executed when the SBC is connected to a target wireless network is written in Python. Python has native commands for sending a string parameter to a terminal shell for execution. Taking advantage of that, many of the automated commands not within the 'Metasploit' stand-alone module, which is later explained, are sent through the native connection between Python and the Terminal. The full flow of the script is depicted in Figure 2. The first thing in the script to be automated was scanning the network with Nmap. The various flags seen below such as '-O', simply specify the types of information that will be needed to be returned into the created file 'recon.nmap'.

Following this, the results in '.nmap' form were converted, shown in Listing 1, into a CSV file through an open source python file called 'nmaptocsv' [10]. As Python is built in with packages to decipher and sort CSV rows and columns into Arrays, having the scanner data saved in CSV form was very important.

```
...
os.system('python nmaptocsv.py -i recon.nmap -f ip-fqdn-
          port-protocol-service-version-os -o recon.csv')
...
```

Listing 1. Sending Nmap output as input to nmaptocsv.py to be converted to CSV.

After sorting the data in 'recon.csv' into various arrays, IP_Array, OS_Array, PORT_Array, etc..., such that index n throughout each array would represent the corresponding bit of information about the (n+1)th machine found on the local network. For example, the information in Index 8 in the OS Array represents the OS of the 9th machine found by Nmap.

After this step, one by one, the IP Addresses of the targets are sent to OpenVAS Management Protocol (OMP). After extensive trial and error, it was decided that sending and receiving data through XML would be easiest to automate. A command showing one of these commands is shown in Listing 2. For OpenVAS, to perform a scan on a target IP and get a report back, a set of processes must be run. The first one of these processes is to set a target by assigning it to a machine IP. The next being to set a task by assigning it to a target, followed by starting the task, and finally getting a report in XML format. Each of these processes returns a UID. In OpenVAS, to refer to the previous data returned by itself, the UID of the previous action must be given back to the system, which in itself can be a daunting task to automate the next command, the SBC would have to know what random UID was assigned after the previous command. To combat this issue, every output given by OMP was piped into a separate XML file, which was promptly parsed for the case-specific UID, and was stored on the SBC for future usage.

```
...
os.system('omp --pretty-print --xml \'<get_tasks/>\' '
          + '> tasks.xml')
...
```

Listing 2. Example in which output XML is sent to 'tasks.xml' to be parsed.

The XML parsing is done through a Python package called 'ElementTree XML.' It specifies in the OMP documentation, where the location of the UID will be for each returned XML file, which is the reason the UID is able to be returned without any problem.

Once the final report XML file is created, as seen in the excerpt shown in Listing 3, per OMP documentation, there will be a 'CVE' tag either with the string 'NOCVE', or a string specifying a CVE the target machine may be susceptible to, such as 'CVE-2007-2447'.

This 'CVE string' is extracted and if the CVE is valid, then it is sent to Metasploit to see if Metasploit has any exploits made for that CVE; if not valid, then the next machine IP address in IP_Array is sent to OpenVas for vulnerability assessment.

```
...
<host>192.168.1.103
    <asset asset_id=""></asset>
</host>
<port>445/tcp</port>
    <nvt oid="1.3.6.1.4.1.25623.1.0.10394">
        <type>nvt</type>
        <name>SMB log in</name>
        <family>Linux</family>
        <cvss_base>0.0</cvss_base>
        <cve>CVE-2007-2447</cve>
</port>
...
```

Listing 3. Sample XML from the returned report for the vulnerability scan.

Automating Metasploit or more specifically 'MSFconsole' was a nontrivial task throughout this process. The problem lies in the fact that when MSFconsole is called from the terminal, and opens its own separate module inside it, the separate module cannot be accessed with the regular means Python provides to access the standard shell. The command in the Python script opening the MSFconsole is shown in Listing 4.

```
...
os.system('MSFconsole')
...
```

Listing 4. Calling MSFconsole from terminal in Python.

Due to this, any following commands sent to the terminal via Python for MSFconsole are sent to the separate module MSFconsole runs in within the terminal. It was possible, we found to pass a resource script (.rc) file in with the command to initialize MSFconsole. However, this option was meant to pass a few unchanging commands in for MSFconsole to run as part of the initializing process. What we found needed to be done for true automation of MSFconsole was to procedurally generate the contents of the .rc file, thereby retaining needed dynamism for exploiting various local networks.

The way we went on by doing that is quite unique and may help anyone in the future needing Python-based automation of MSFconsole. While other researchers have stumbled upon very similar solutions as we did, ours can work in a variety of different situations, rather than in static situations that only work if a known exploit is manually entered in. Metasploit's native language is Ruby, and therefore has more support for Ruby Automation than Python Automation. In fact, within the .rc file that can be passed in with MSFconsole, there is the option to create code blocks of Ruby code to achieve automated tasks. However, this standalone feature cannot integrate and pass variables with outside Ruby scripts and therefore will not allow the needed transition back and forth between spawning commands within MSFconsole and to the root shell. Nevertheless, we are using Python, and found that we could, within the main Python script that every tool is automated through, open and write commands to the .rc file when needed, only invoking MSFconsole with its corresponding resource script file after we have finished generating all the commands it must run for the entirety of the attempted exploitation session of all the machines on the local network and closed the file. This process is shown in Listing 5.

Going back to the transition between OMP and Metasploit, as OMP passes the CVEs that the target IP address maybe susceptible to MSFconsole, MSFconsole runs a series of searches, using commands written to the aforementioned resource script file, to find out if it has any preexisting exploits made for each CVE given to it. If it does not find any existing exploits for any of the CVEs given to it, control will pass back to OMP to start scanning the next machine in the network as provided by the IP_Array. If it does find existing exploits, MSFconsole will sequentially execute said exploits against the target machine. Again, if none of the exploits work, and a shell is not spawned, control is passed back to OMP to begin scanning for vulnerabilities on the next machine. If in the case that the exploit works, and a shell is spawned, another thread will be opened, leaving the current untouched, and the next machine will go through the whole process again in that thread.

```
...
msf = open("generated.rc","w")
...
# Logic that leads to following commands
...
msf.write("\n" + "set␣RHOSTS␣" + IP_Array[x])
msf.write("\n" + "set␣RHOST␣" + IP_Array[x])
msf.write("\n" + "exploit")
...
# After all logic has been passed and file is fully written
msf.close()
os.system('MSFconsole␣-r␣generated.rc')
...
```

Listing 5. Sample Commands sent to the .rc file and run at the end of file generation through logic using the '-r' flag.

IV. Experiments

Trials ranged between networks with only a single target machine to networks with multiple target machines. The local networks with multiple target machines consisted of 3 vulnerable VM's, as well as 2 machines that were not expected to be exploitable. All the trials were conducted on local networks consisting of machines with OS images known to be susceptible to some exploits on Metasploit. For example, we used the 'Metasploitable' VM [9] in VirtualBox [13] as our 'go-to' exploitable machine in the testing phases. The 'pwnOs' VM was also used as a VM for one machine on the local network. On various machines, the Metasploitable VM was set up and connected to the target local network and the script on the SBC, which in our case was the ODROID XU4, was executed. First, the script scanned the network and sorted the information into a CSV file. Behind the scenes, the data from the CSV file was sorted into various arrays for later extraction.

```
IP;FQDN;PORT;PROTOCOL;SERVICE;VERSION;OS
192.168.0.1;;21;tcp;ftp;;Linux 2.6.23 - 2.6.38
...
192.168.0.101;;88;tcp;kerberos-sec;;Apple Mac OS X 10.7.0
...
```

Listing 6. CSV output after Nmap scan. This data is sorted into various arrays.

Next, in this trial, the script ran as expected into OpenVas and scanned each IP address and behind the scenes, the responses were parsed for the UID assigned at the end of each command as in shown Listing 7.

```
Starting OpenVas Services
<create_target_response status_text="Target␣exists␣already" status="400"
<create_task_response id="15aece0b6328-4f1d931e-7aaf455cfe52" status␣
<start_task_response status_text="0K,␣request␣submitted" status="202">
<report_id>fd7aaea52a554d53-9d47c7d1f45a3113<lreport_id>
</start_task_response>
<create_target_response status_text="Target␣exists␣already" status="400
<create_task_response␣id="8f2e8088e7b8-4584b877-3565a5def3d8"␣status␣
<start_task_response␣status_text="0K, request submitted"␣status="202">
<report_id>699a33@afcee4544-90efa219e228878e<lreport_id>
</start_task_response>
```

Listing 7. Response of OpenVAS.

Any CVEs found in the OpenVAS report were successfully passed on to MSFconsole and if MSFconsole found a working exploit for it, it set the Target to the IP Address that was the target of the vulnerability scan and successfully attempts to exploit it. For example, Listing 8 is the terminal output while exploiting one machine, it fails trying one exploit, then succeeds after trying another.

```
...
MSFconsole > resource (generated.rc )> spool cve0.txt
Spooling to file cve0.txt...
MSFconsole > resource (generated.rc) > search CVE-2007-2447
MSFconsole > resource (generated.rc) > use exploit/multi/samba/usermap_script
MSFconsole > resource (generated.rc) > set RHOSTS 192.168.0.103
RHOSTS => 192.168.0.103
MSFconsole > resource (generated.rc) > set RHOST 192.168.0.103
RHOST => 192.168.0.103
MSFconsole > resource (generated.rc) > exploit
Handler failed to bind to 192.168.0.102:4444:-
Started reverse TCP double handler on 0.0.0.0:4444
...
```

Listing 8. Automated MSFconsole response and exploitation output to terminal.

We found that out of a local network of 3 vulnerable machines and 2 machines that were not expected to be

vulnerable to any exploit in the metasploit database, 1 to 2 machines consistently had a root shell spawned and could be deemed 'successfully exploited'. In all cases, however, the unique automation techniques that Cybergrenade mapped out worked to the best of their abilities and delivered the right commands at the right times. Failure to exploit any given machine mainly stemmed from either OpenVas not being able to find any CVEs and from Metasploit not being able to match a working exploit with those CVEs.

V. Conclusion

In this paper, we introduced Cybergrenade, which is a framework that automates Nmap, OpenVAS, and Metasploit as part of the exploitation process all underneath one Python Script on an SBC. SBCs are quickly becoming more and more widespread and its popularity shows no signs of slowing down. Many have used them as covert pentesting tool-kits before, but they stop short in completely automating the exploitation process. The automation has many benefits that stay true to the covert nature of exploitation. This covert behavior bids well to the whole point of using a SBC rather than a larger computer, to remain undetected. The flexibility automation provides opens up endless possibilities previously unrealizable without it.

Two unique contributions in this paper, that can be used in other projects, are the successful automation of OpenVAS/OMP in the manner shown, and the specific way Cybergrenade provides flexible automation of Metasploit/MSFconsole through Python. As this project successfully deals with the unpredictability OMP's UIDs pose in the automation of the tool, further research and usage of this technique can allow anyone who needs a free vulnerability scanner like OpenVAS automated a path to do so. Also, this paper presents a good way forward for any group looking to automate MSFconsole using Python by writing to the resource script file whenever needed and only initializing MSFconsole once the .rc file has been fully generated. As the blueprint for automated MSFconsole has been laid out, it can be followed to allow even smarter exploitations in the future as most of what a human can do manually on MSFconsole can now be automated as predictability allows. Automated exploitation of local networks is most practical in tandem with SBCs due to the covert nature of SBCs, however automated solutions as presented in this paper are also viable on any Linux-based Computer that can use Nmap, OpenVAS, and Metasploit.

Acknowledgment

This work was partly supported by the US National Science Foundation Research Experiences for Undergraduates Sites Program Grant No: REU-CNS-1461119 and US NSF-CAREER-CNS-1453647. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the funding agencies.

References

[1] K. Scarfone, M. Souppaya, A. Cody, and A. Orebaugh, "Technical guide to information security testing and assessment," National Institute of Standards and Technology, Tech. Rep., 2008.

[2] F. Palmieri, U. Fiore, and A. Castiglione, "Automatic security assessment for next generation wireless mobile networks," *Mobile Information Systems*, vol. 7, no. 3, pp. 217–239, 2011.

[3] F. Holik, J. Horalek, O. Marik, S. Neradova, and S. Zitta, "Effective penetration testing with metasploit framework and methodologies," in *2014 IEEE 15th International Symposium on Computational Intelligence and Informatics (CINTI)*, Nov. 2014, pp. 237–242. DOI: 10.1109/CINTI.2014.7028682.

[4] L. Epling, B. Hinkel, and Y. Hu, "Penetration testing in a box," in *Proceedings of the 2015 Information Security Curriculum Development Conference*, ser. InfoSec '15, Kennesaw, Georgia: ACM, 2015, 6:1–6:4, ISBN: 978-1-4503-4049-6. DOI: 10.1145/2885990.2885996. [Online]. Available: http://doi.acm.org/10.1145/2885990.2885996.

[5] J. Muniz, *Penetration testing with raspberry pi*. Packt, 2015.

[6] V. Tilemachos and C. Manifavas, "An automated network intrusion process and countermeasures," in *Proceedings of the 19th Panhellenic Conference on Informatics*, ser. PCI '15, Athens, Greece: ACM, 2015, pp. 156–160, ISBN: 978-1-4503-3551-5. DOI: 10.1145/2801948.2802001. [Online]. Available: http://doi.acm.org/10.1145/2801948.2802001.

[7] Y. Hu, D. Sulek, A. Carella, J. Cox, A. Frame, and K. Cipriano, "Employing miniaturized computers for distributed vulnerability assessment," in *11th International Conference for Internet Technology and Secured Transactions (ICITST)*, Dec. 2016, pp. 57–61. DOI: 10.1109/ICITST.2016.7856666.

[8] (2017). Kali linux, [Online]. Available: https://www.kali.org/kali-linux-documentation/.

[9] (2017). Metasploitable3, [Online]. Available: https://github.com/rapid7/metasploitable3.

[10] (2017). Nmaptocsv, [Online]. Available: https://www.github.com/maaaz/nmaptocsv.

[11] (2017). Odroid xu4, [Online]. Available: http://odroid.com/dokuwiki/doku.php?id=en:odroid-xu4.

[12] (2017). Omp documentation, [Online]. Available: http://www.openvas.org/omp-6-0.html.

[13] (2017). Oracle virtualbox, [Online]. Available: https://www.virtualbox.org/.

[14] (2017). Raspberry pi 3b, [Online]. Available: https://www.raspberrypi.org/products/raspberry-pi-3-model-b/.