Global Information Assurance Certification Paper

Secure Design with Exploit Infusion

GIAC (GCIH) Gold Certification

Author: Wen Chinn Yew, wenchinn@outlook.com
Advisor: Daniel Lyon

Accepted: November 6th 2014

Abstract

This paper introduces the concept of Exploit Driven Development for secure software. It presents traditional principles of secure design and suggests how adversaries' exploits and techniques can be used to approach and augment a secure development process. The paper recommends writing code to thwart an exploit. Exploit Driven Development aims to reduce the cost of software development and instill a higher level of security in products.

1. Introduction

In the age of a highly digitally connected world, the ever-increasing security threat has prompted many initiatives to address it. One important area is to build security into software development. An example is the "Build Security In" project by the Department of Homeland Security (DHS, 2014). The project hosts a website, in a collaborative effort, to provide useful resources for the software development community to tap on and build security into every phase of software development.

Microsoft's "The Trustworthy Computing Security Development Lifecycle" is an illustration of including security-focused activities and deliverables to each software development phase. There is a significant drop in the number of post versus pre Security Development Lifecycle (SDL) security bulletins for Microsoft's software releases. The SDL process includes "the development of threat models during software design, the use of static analysis code-scanning tools during implementation, and the conduct of code reviews and security testing" (Lipner & Howard, 2005).

Threat modeling and risk assessment during design helps to build security into software. Military strategist Sun Tzu, author of an ancient Chinese book on military strategy, said that one must know the enemy as well as the self in order to win battles (Lionel, 2007). When the threats and vulnerabilities are known, mitigation work can be carried out more effectively. Exploit Driven Development (EDD), a parallel drawn from Test Driven Development (TDD) (Beck, 2003), can play a part in secure software development. TDD is about writing a test before writing just enough production code to fulfill that test. EDD is to write code to pass an exploit test.

Test Driven Development has its fair share of supporters and detractors. This paper is not advocating which development methodology to use; rather it draws upon the idea of designing with intent to pass a penetration test. The primary intent of the code is to perform a certain function and the secondary intent is to pass exploit tests.

Wen Chinn Yew, wenchinn@outlook.com

2. Principles of secure design

The classic paper "The Protection of Information in Computer Systems" (Saltzer & Schroeder, 1975) introduced eight design principles: Economy of Mechanism, Fail-safe defaults, Complete mediation, Open design, Separation of privilege, Least privilege, Least common mechanism and Psychological acceptability. These principles are still widely acknowledged to be very relevant now (Smith, 2012). Adding to these eight principles are four more current secure design principles advocated: Securing the weakest link, Defense in Depth, Reluctance to Trust, Promoting Privacy (Viega & McGraw, 2002). Let's take a look at each of them in more detail below.

2.1. Economy of Mechanism

According to Saltzer and Schroeder (1975), a design should be simple and small. It is easier to check and test a functionality that has a simple execution path compared to one with many alternate paths and interactions. Viega and McGraw (2002, p.104) suggest to "Keep It Simple" because complex design is hard to understand and maintain. Problems might be overlooked during analysis of a complex system.

2.2. Fail-safe defaults

When a system fails, it should do so in a secure manner. A system should be locked down by default and every access requires a request to be made. Any mistakes in implementation will then fail with a denial to access. This is considered a safer method compared to the contrary where mistakes result in access rights.

2.3. Complete mediation

The design principle of "Complete mediation" requires that access to every object must be checked for authorization. It implicitly states that there must be a method devised to identify and verify an access request. This method should not be circumvented by any means and demands a thorough system-level view and design. Saltzer (1975) also highlighted careful handling of caching permissions.

2.4. Open design

Security through obscurity is what "Open design" is intended to avoid. Insider threats or reverse engineering can compromise obscurity. In addition, an open design can be scrutinized by a larger community to harden it. An example is in the field of cryptography where an open algorithm can invite constructive feedback on weaknesses. The failure of the CSS (Content Scramble System) DVD copy protection can be attributed to a closed design that depended on obscurity (D. Miller, 2005, p.91).

2.5. Separation of privilege

A protection mechanism that requires two checks is more secure than one that only requires a single check. An analogy is in money cheque signing when two signatures from two individuals are necessary to authorize a withdrawal.

2.6. Least privilege

The idea of "Least privilege' is to allocate the minimal set of privileges necessary to complete the job. This will reduce exposure of the system to threats should an error or incident occur. It is also the reason sandboxing is used to separate running programs in a computer for example. An untested or untrusted program can be sandboxed so that its impact to other programs is limited.

2.7. Least common mechanism

Different components should avoid using the same mechanism to access a resource. Examples of a shared mechanism are a shared variable, state or method. Saltzer and Schroeder (1975) pointed out that every shared mechanism represents a potential information path between users and errors in that mechanism might affect the different users. Limiting the sharing of mechanisms can also help reduce design complexity and consideration of interaction between users.

2.8. Psychological acceptability

The usage of a security mechanism should not be too much of a burden on a user. A high burden imposed by security constraints discourages use of the system or encourages circumvention.

2.9. Securing the weakest link

In a system made up of many components, it is easier to penetrate the system by choosing a component that has the weakest security built in. One example of the path of least resistance is, adversaries chose to target end points of a communication channel before/after the information is encrypted/decrypted rather than try to defeat the cryptography used in the encrypted channel.

2.10. Defense in Depth

The idea behind "Defense in Depth" is to provide layered defense so that if one layer fails, there is still another layer of protection. Much like peeling an onion, attackers need to penetrate through different layers of the onion before it can reach the core.

2.11. Reluctance to Trust

The environment a system is in should be considered insecure as a default. This means assuming inputs to a system can be from a foe and with evil intent, or even a foe masquerading as a friend. When the developer understands these threats, the system can be designed to react appropriately to attacks.

2.12. Promoting Privacy

Protecting systems from revealing private information is critical. These information can often empower adversaries to launch attacks on the system or other systems.

3. Exploits and techniques

Having discussed the principles of secure design, it is now appropriate to take an opposite view, from that of an adversary. Hacker's exploits and techniques can be categorized into five stages: Reconnaissance, Scanning, Gaining Access, Maintaining Access and Covering the Tracks (SANS, 2014). Simply put, it is first to gather useful information from different sources and next to perform actual pre-attack to identify vulnerabilities. The third step is to perform the attack to gain a foothold on the system. Fourth is to maintain this foothold and finally, to erase traces that the system is compromised.

Reconnaissance from public databases or registration bodies, website searches, and intelligence gathering tools such as Maltego can produce information useful in Social Engineering or Scanning (SANS, 2014). Scanning is a procedure of identifying weaknesses in a system that creates opportunities for exploits. Well known scanning terminologies are War Dialing (to locate modems), War Driving (to search for Wi-Fi wireless networks), Network Mapping (to map out the physical connectivity of networks) and Port Scanning (to identify open ports, type of network system and even applications). It is also possible to know the Operating System used by detailed observation of how different systems react to the same request. This technique is known as fingerprinting. Examples of vulnerability scanning tools for the network are Nessus and OpenVAS and Nikto for the web.

For Gaining Access, attackers have many techniques which are constantly evolving with creativity. This paper will highlight some basic methodologies. Buffer overflow is a classic example. It is the result of moving data without checking on the size, thereby overwriting adjacent memory. A buffer overflow vulnerability might allow malicious code to be injected as part of the unchecked data and executed on a system. Another example is the use of a format string attack. Unchecked user input and errors in format string usage can result in similar malicious code execution.

To illustrate some other methods, let's look at the different communication layers of the Internet (IETF, 1989). They are the Data Link, Internet Protocol, Transport and Application layers. The Data Link layer involves the hardware MAC address and the Internet Protocol layer, the IP address. These addresses can be spoofed. If controls are based on these relationships, they can be subverted.

The Transport layer (TCP/UDP) is another target. For example, creating many incomplete TCP handshakes can result in a Denial of Service (DoS) attack. A 3-way TCP handshake mechanism is required to establish a valid connection between two parties. If one party creates a large number of incomplete handshakes, the other party might suffer a memory crash if its implementation is not robust enough to anticipate this attack. It is also possible to alter the contents of the TCP packet such as source address and sequence number to masquerade as a legitimate user.

At the Application layer, session hijacking is one technique employed to steal an authorized session. This can be done by masquerading as a legitimate user of an existing session, or intercepting communication between the origin and destination and modifying the content to both.

It is also worthy to mention Password Cracking as another exploit. Besides brute force guessing, there are dictionary attacks with a list of possible passwords, hybrid attacks that build on a dictionary attack by appending characters, and rainbow table attacks with pre-generated password hashes. With hashes pre-generated, one can simply compare the hashes for a match.

Having gained access to a system, it is important to also be Maintaining Access and Covering the Tracks. Attackers plant backdoors and Rootkits into a system to enable them to bypass normal access controls to facilitate re-entry. They also like to cover their tracks by altering system logs and hiding any artifacts such as files or information about abnormal activities that will betray their existence.

4. Secure design with exploit infusion

Combining both perspectives of a defender and an attacker will help to create a secure design. In a typical development lifecycle, the different phases are normally Requirements, Design, Implementation, Test, Integration and System Test, with Reviews interspersed at appropriate phases (Sommerville, 2011).

For a security focused development lifecycle, security requirements and risk analysis need to be considered during Requirements. Secure design principles and threat modeling can be applied during Design. In the Implementation phase, actual code will be written. Code reviews, static analysis tool scans and penetration testing are normally performed only after the Implementation phase. This means that a rework is required to rectify problems uncovered after implementation.

By introducing the concept of Exploit Driven Development, this paper suggests to instill the idea of writing code to thwart an exploit. When designing a function, the intent is to fulfill its primary requirement and to keep in mind secure design principles. A secondary goal is to write exploit test cases for the function and to write code to pass the test. In Test Driven Development, Kent's (2009, p.9) rule is "Don't write a line of new code unless you first have a failing automated test". The TDD's mantra of Red/Green/Refactor, where Red is to fail and Green is to pass, requires repeated steps of adding a test case that fails and then writing code to pass and then finally refactoring. A test case that is an exploit will result in code that will pass the exploit. It is like performing penetration testing in the early phase of development. The result is that in the end-phase of development, the defect density of code will be reduced. This can help minimize rework that will be incurred near the end of development which is typically more costly (McConnell, 1996).

In order to help identify what types of exploit testing can be employed during design and coding, it is useful to enumerate through the 12 secure design principles presented in Section 2 and consider which of the exploit phases discussed in Section 3 can cause the design principle to fail.

Wen Chinn Yew, wenchinn@outlook.com

Table 1 below presents such a map of traditional principles of secure design versus relevant exploit phases against it. This table can serve as a guideline to influence design and development and is discussed in more detail below.

Table 1 - Exploit Phases against Principles of Secure Design

	Reconnaissance	Scanning	Gaining Access	Maintain Access/ Covering Tracks
Economy of Mechanism			√	
Fail-safe defaults			√	
Complete Mediation			√	√
Open design	√		√	
Separation of Privilege			√	√
Least Privilege			√	√
Least Common Mechanism			√	
Securing the Weakest Link	√	√		
Defense in Depth			√	
Reluctance to Trust			√	
Promoting Privacy	√	√	√	

4.1. Reconnaissance

Reconnaissance is about looking up information and searching to uncover information. Though the Social Engineering and Scanning aspect might not be directly applicable to design and coding, the idea of looking, searching and checking can be extended to test against security through obscurity. A secure design should not depend on hiding information. It should use established secure methods if available instead of reinventing one. A test in this case might be in the form of a checklist or review (Myers, 2004).

Flaws in the use of cryptography can be a weak link in design. A method of testing for implementation or algorithm issues in cryptography is to use cryptanalysis. Some examples of cryptanalysis are chosen plain text/cipher text methods where a tester inputs any plain text/cipher text and analyzes the corresponding outputs to discover

weaknesses (Katz & Lindell, 2007, p.8). A poorly generated initialization vector, salt or mistakes in coding might be detected by checking the output against the same input. Validating the output of a component against the same input can also uncover weaknesses in block cipher mode. This is illustrated famously in AES-ECB (Advanced Encryption Standard – Electronic Codebook) mode encryption where an image of a Penguin after encryption still shows its silhouette.

Diagnostic and debugging information, system statuses and responses are items that can be used to infer something about a system. These will prove invaluable in reverse engineering. A case in point is in a web response to a user name and password entry. If a web response to a valid user name and invalid password is to indicate that the user name is right but password is wrong, it exposes the presence of the valid user name. This response reduces the scope for an attacker who now knows the user name. A better response is a general message notification that the authentication failed (OWASP, 2008, p.113). In this example, tests can be written to check that a single general response is returned for different failure scenarios.

4.2. Scanning

Vulnerability scanners used during penetration testing can be used early in development to discover weak links. The applicability however is dependent on the functional completeness of the developed component under test. Unit test cases need to be developed when third party tools cannot be used. An example is to test that only allowed listed ports are opened or only allowed services are loaded.

Whitelisting of allowed ports or services and for data validation in general, is a preferred method compared to blacklisting (OWASP, 2013). The reason is that there might be combinations or permutations that could be omitted while coming up with a blacklist. When a developer is implementing a whitelist of ports, he can create a single function that can be called to check for allowed ports. However if a developer is integrating code from a third party, it is more difficult to verify the allowed ports through code reviews. This is especially so if the used port numbers are scattered throughout the code. A test method in this case is to enumerate through all possible port numbers to connect on the system, much like a scanner.

Scanning tools are able to accomplish extensive and varied tests. A unit test case on the other hand can be focused and directed. This is because it is created by the developer with specific information about the system. For example a scanner might depend on a pre-known list of items to check against and report whether it is present or absent. The scanner is unable to report on items not on the list. The developer understands the system and can check to ensure only minimal required items are present.

4.3. Gaining Access

In Table 1, many principles of secure design can be challenged to check a component's security robustness. In "Fail-safe defaults", it is important to validate that access to a valued asset is not available by default. In "Complete mediation", access permissions granted to an object should not be cached. "Separation of privilege" recommends that access be granted based on the success of more than one condition. These can result in writing tests to ensure an out of box or reset situation will remove access to valued assets. For example, in the implementation of a Web logout function, it is essential to test that all session tokens are made invalid and sensitive data are not cached. This will help prevent a "replay" attack where an attacker presents again the valid data (OWASP, 2008, p.133).

To test whether components adhere to the principle of "Least Privilege", an example can be found in the Data base listener of Oracle databases. OWASP (2008, p.85) states that it is important "to give the listener least privilege so it cannot read or write files in the database or in the server memory address space". So test cases should be crafted to confirm that the component is unable to perform actions outside of its expected scope.

The topic of "Reluctance to Trust" presents more interesting discussion points. An important aspect of an exploit test that can be employed here is that of Fuzzing and Parameterized Unit Test. Fuzzing involves presenting invalid, unexpected, or random input data to a component (B.P. Miller, 2008). It is commonly used in tools to test for security vulnerabilities in systems. They typically are file- or protocol-based at the system level. Parameterized unit tests (PUTs) are tests that take parameters (Tillmann & Schulte, 2005). Exploit tests can be implemented on components with the aim to

parameterize/fuzz. These tests exploit buffer overflow, format string vulnerabilities and insufficient input validation in codes. Existing unit test automation can be leveraged to reduce manual test burden. Developers might already be used to performing boundary tests on components, and those tests can be extended further to include fuzzed data. Fuzzed data can be produced with automatic data generation methods or by manipulating or mutating valid data. Fuzzing at the unit test level, where the developer has access to source code and is aware of the design, may at times be able to identify weaknesses faster than a third party tool that may spend time fuzzing parameters that are out of scope.

Masquerading as an authorized agent is a common technique used to illegally gain trust. From protocol to application level, any predictability in sequence number, session identity number or initialization vector can be a weakness. If an attacker knows what will be the next sequence number of a TCP packet, he can inject a crafted packet that appears legitimate to the system. Similarly if a server depended upon a session identity number to identify an authenticated client, an attacker can bypass the authentication schema of an application. When an initialization vector used in a cryptographic algorithm can be predicted, it can aid in a chosen plain text attack by comparing the output of the algorithm to know the input (Katz & Lindell, 2007, p.82). Since random numbers play a very important role in cryptography and security, an exploit test should be made to check for the degree of randomness. The art of testing for randomness is not trivial. An industry-standard test suite from the National Institute of Standards and Technology (NIST) can be considered (NIST, 2010).

4.4. Maintaining Access/Covering the Tracks

In the final two exploit phases of "Maintaining Access" and "Covering the Tracks", it is helpful to ensure good adherence to "Complete mediation" and "Least Privilege". Always check a request for access rights and do not give components excessive privileges. Imagine the case when an adversary gained access to a component, and then makes a move to another component to hide its trail, all because the former gave it privileged access to the latter. To give a simple example, an attacker compromised Machine A with a malware A. Through Machine A, the attacker gains illegitimate access to Machine B. The attacker then plants a different malware variant B into Machine B. In

the event that malware A is detected, B might still remain undetected. Other tests at a higher level can be testing the defense mechanisms in audit and logging.

Wen Chinn Yew, wenchinn@outlook.com

4.5. Exploit test concept during development

To reinforce the concept of exploit testing in the implementation phase of development, this Section will first show a typical development process flow. The TDD/EDD methodology and its infusion into development will then be illustrated. Figure 1 below is a typical Waterfall Model (Sommerville, 2011).

Figure 1

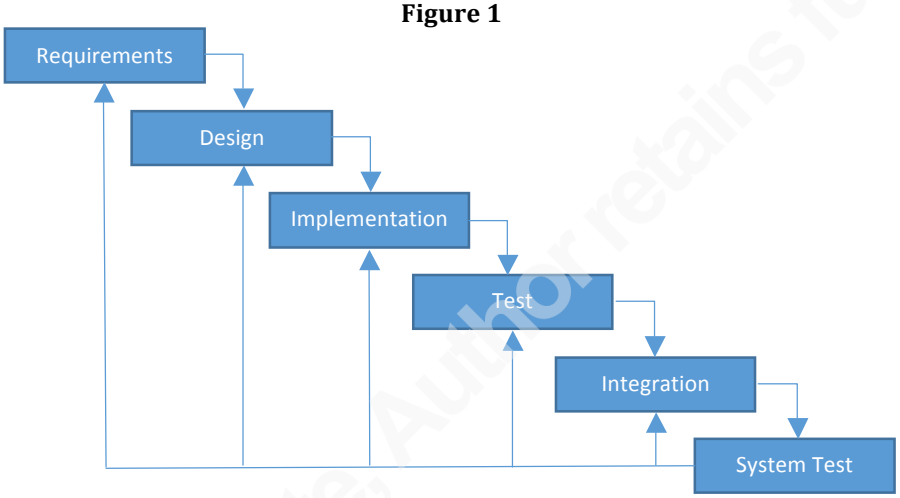
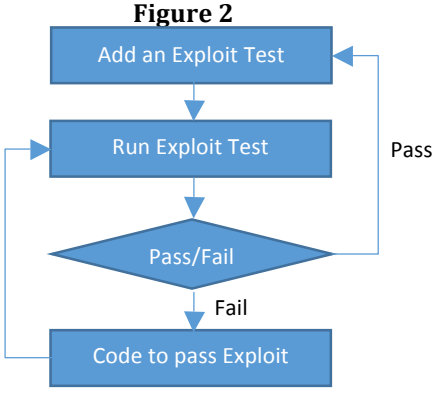


Figure 2 is a flowchart of typical steps in TDD (Ambler, 2013). The first step is to add an identified exploit test case and the second step is to execute the test case. Typically on the first pass, the code will fail the test case because of a lack of coding. In the third step, the developer will write code to address the test case. The test is then executed again and when it passes, the cycle can repeat with the addition of another exploit test case.

Figure 2

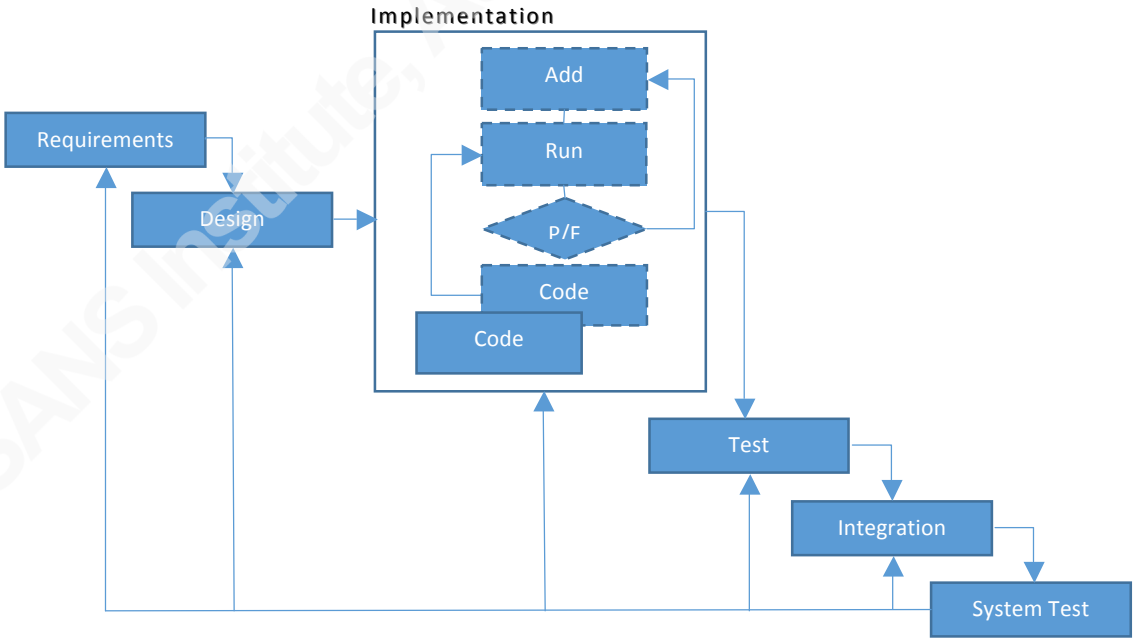


Wen Chinn Yew, wenchinn@outlook.com

TDD/EDD methodology implicitly requires that a developer begins by identifying the different possible exploits that are relevant to the component under test. This is necessary for the developer to know what test case to add. It will complement threat modeling that was carried out earlier during the design phase. The code that is written to address the test case can be easily identified as code that might be important during a security code review. Tests are executed early during development since they are conducted in the implementation phase.

Figure 3 shows where Exploit Driven Development comes in during the development process. It must be noted that the Waterfall Model is chosen just as an example and other development models can also apply the same exploit test concept during coding.

Figure 3



Although the emphasis is on exploit test during coding, it must be stressed that sometimes unit or integration level testing makes more sense. Exploit testing should then be carried out at that level. Exploit tests that subject the component to extreme loading conditions (also known as stress testing) are also often able to identify the Achilles Heel of buffer, memory, computation bandwidth or timing issues (Dustin, Rashka & Paul,

2004, p.41, 248). All these in summary serve to complement test strategies normally already employed.

4.5.1. Exploit Driven Development – An Example

An Authentication and Authorization Access (AAA) function that loads a requested service is used as an example to show how to apply EDD. Pseudo code (not functionally complete) is used for ease of illustration. The function has three input parameters: user name, password and requested service. It compares the user name/password pair against an encrypted list of user name/password pairs for a match. When there is a match, the function will load the requested service. There are requirements for valid inputs and allowed services.

Three possible candidates for exploits are the input parameters, the access to the encrypted file and the service to load. The test cases to execute are, invalid input parameters, illegal access to the encrypted file, illegal access to the list of user name/password pairs and the loading of an illegal service.

The first exploit test case is to subject the function to invalid input parameters. Invalid input is created as entries in a list and an automated test calls the AAA function, cycling through the list of inputs. AAA should deny permission in order to pass the test. The invalid input list can be automatically generated with a test automation tool, such as Robot Framework's string library (Robot Framework, 2014), to produce fuzzed data. The example starts off with an empty AAA function that always grants permission. Note that it is not normally the case to have non-functional code before EDD can be applied. This will be shown later in the Section.

Test Case 1: Call AuthenAuthorizeAcess function with invalid inputs

```
BOOL AuthenAuthorizeAccess(UserName, Pwd, Svc){
      return ALLOW
}
```

Execute the test and it will fail. Next, write code (in bold below) to make the test pass. As a general guideline, the code should do the opposite of what the test case is testing against. Test case 1 is testing for invalid inputs, so the code should perform checks for valid inputs.

```
BOOL AuthenAuthorizeAccess(UserName, Pwd, Svc){
      if ValidateInputs(UserName, Pwd, Svc)
            return ALLOW
      else
            return DENY
}
BOOL ValidateInputs(UserName, Pwd, Svc){
      if Inputs are of allowed character set AND length
            return TRUE
      else
            return FALSE
}
```

Run the test again and it will now pass. In order to pass the test of invalid inputs, code has to be written to address the exploit. During development, the code will also be checked against functional test cases. Positive and negative (where exploit test is a subset) test coverage will improve the quality of the code.

Now, extend the functionality of AAA (in italics) to access the encrypted file.

```
BOOL AuthenAuthorizeAccess(UserName, Pwd, Svc){
      if ValidateInputs(UserName, Pwd, Svc)
             if AccessFile()
                  return ALLOW
            else
                  return DENY
      else
            return DENY
}
BOOL  AccessFile(){
      Access File as Read
      Read File, Decrypt File, Obtain UsernamePwdList
      if Above successful return TRUE else return FALSE
}
```

The second exploit test case (in bold below) is to access the encrypted file as writable. AAA must deny permission to pass the exploit.

Test Case 2: Access the encrypted file as writable

```
BOOL  AccessFile(){
#if EDDTEST_2
      Access File as Write
#else
      Access File as Read
#endif
      Read File, Decrypt File, Obtain UsernamePwdList
      if Above successful return TRUE else return FALSE
}
```

Execute test case 2. The test fails. Write code (in bold below) to pass the test.

```
BOOL  AccessFile(){
#if EDDTEST_2
      Access File as Write
#else
      Access File as Read
#endif
      if File access NOT ReadOnly
            LOG File access writable
            return FALSE
      Read File, Decrypt File, Obtain UsernamePwdList
      if Above successful return TRUE else return FALSE
}
```

Re-run the test and it will pass. Note that code is written to address the exploit.

Extend the functionality of AAA (in italics) further to authenticate and authorize an access by matching the input user name/password pair to UsernamePwdList in the encrypted file. When a match exists, load the requested service.

```
BOOL  AccessFile(){
#if EDDTEST_2
      Access File as Write
#else
      Access File as Read
#endif
      if File access NOT ReadOnly
            LOG File access writable
            return FALSE
      Read File, Decrypt File, Obtain UsernamePwdList
      if Input pair Match entry in UsernamePwdList, Load Service
      if Above successful return TRUE else return FALSE
}
```

The third exploit test case (in bold below) is to access the UsernamePwdList after it is not required and AAA must deny permission.

Test Case 3: Access UsernamePwdList after it is not required

```
BOOL  AccessFile(){
#if EDDTEST_2
      Access File as Write
#else
      Access File as Read
#endif
      if File access NOT ReadOnly
            LOG File access writable
            return FALSE
      Read File, Decrypt File, Obtain UsernamePwdList
      if Input pair Match entry in UsernamePwdList, Load Service
#if EDDTEST_3
      if UsernamePwdList NOT EMPTY
            return TRUE
      else
```

```
            LOG UsernamePwdList not cleared
            return FALSE
#endif
      if Above successful return TRUE else return FALSE
}
```

The test did not pass. Write code (in bold below) so that the test can pass.

```
BOOL  AccessFile(){
#if EDDTEST_2
      Access File as Write
#else
      Access File as Read
#endif
      if File access NOT ReadOnly
            LOG File access writable
            return FALSE
      Read File, Decrypt File, Obtain UsernamePwdList
      if Input pair Match entry in UsernamePwdList, Load Service
      Clear UsernamePwdList
#if EDDTEST_3
      if UsernamePwdList NOT EMPTY
            return TRUE
      else
            LOG UsernamePwdList not cleared
            return FALSE
#endif
      if Above successful return TRUE else return FALSE
}
```

The test is successful. Note again that code is written to address the exploit.

The fourth and last exploit test case is to load a disallowed service. In this example, it is assumed that "Load Service" will load any services. The function AAA should deny permission for the test to pass.

Test Case 4: Call AuthenAuthorizeAccess with disallowed service

Execute the test and it will fail. Next, add code (in bold below) to pass the test.

```
BOOL  AccessFile(){
#if EDDTEST_2
      Access File as Write
#else
      Access File as Read
#endif
      if File access NOT ReadOnly
            LOG File access writable
            return FALSE
      Read File, Decrypt File, UsernamePwdList
      if Input pair Match entry in UsernamePwdList
            if RequestedServiceAllowed()
                  Load Service
```

```
            else
                  LOG Disallowed service
                  return FALSE
      Clear UsernamePwdList
#if EDDTEST_3
      if UsernamePwdList NOT EMPTY
            return TRUE
      else
            LOG UsernamePwdList not cleared
            return FALSE
#endif
      if Above successful return TRUE else return FALSE
}
```

Re-execute the test and it will now pass. The test ensures that code is written to check for allowed services.

The four test cases above illustrate how to apply EDD in practice. A general guideline is to add a test case so that the functionality of the component will fail, and then add code to do the opposite of what the test case is testing against.

5. Conclusion

This paper presented the concept of adding exploit tests during the implementation phase of a software development. This approach aims to reduce the cost of development by requiring security design thoughts and tests be executed earlier to decrease the amount of rework towards the end of a development. It also hopes to instill a higher level of security in products by not depending only on final system tests to uncover vulnerabilities.

In defining the methodology to identify the different exploits that are applicable to a component, this paper proposes a look at the principles of secure design versus adversaries' exploits and techniques as guidance. A few examples are given for each exploit phase as to what to test for each design principle. The test might be in the form of a checklist, a review, test code, test tools, etc. It also highlights the importance of automation in the tests.

Wen Chinn Yew, wenchinn@outlook.com

6. References

Ambler, S.W. (2013). Introduction to Test Driven Development (TDD). Retrieved 8 Sep 2014 from
http://agiledata.org/essays/tdd.html

Beck, K. (2003). Test Driven Development By Example. Addison-Wesley

Department of Homeland Security (2014). Build Security In. Retrieved Jul 30, 2014 from https://buildsecurityin.us-cert.gov/

Dustin, E., Rashka, J. & Paul, J. (2004). Automated Software Testing: Introduction, Management, and Performance. Addison-Wesley

Giles, L. (2007). The Art of War by Sun Tzu

IETF (1989). Requirements for Internet Hosts – Communication Layers. Retrieved Jul 30, 2014 from
http://tools.ietf.org/html/rfc1122

Katz, J. & Lindell, Y. (2007). Introduction to Modern Cryptography. CRC Press

Lipner, S. & Howard, M. (2005). The Trustworthy Computing Security Development Lifecycle. Retrieved Jul 30, 2014 from
http://msdn.microsoft.com/en-us/library/ms995349.aspx

McConnell, S. (1996). Software Quality at Top Speed. Retrieved Sep 8, 2014 from http://www.stevemcconnell.com/articles/art04.htm

Wen Chinn Yew, wenchinn@outlook.com

Miller, B.P. (2008). Fuzz Testing of Application Reliability. University of Wisconsin-Madison. Retrieved 8 Sep, 2014 from http://pages.cs.wisc.edu/~bart/fuzz/

Miller, D. (2005). Black Hat Physical Device Security: Exploiting Hardware and Software. Syngress Publishing

Myers, G.J. (2004). The Art of Software Testing, Second Edition. John Wiley & Sons, Inc

NIST (2010). A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications, Special Publication 800-22 Revision 1a. Retrieved Jul 30, 2014 from http://csrc.nist.gov/publications/nistpubs/800-22-rev1a/SP800-22rev1a.pdf

OWASP (2008). OWASP Testing Guide V3.0. Retrieved Jul 30, 2014 from http://www.owasp.org/images/5/56/OWASP_Testing_Guide_v3.pdf

OWASP (2013). Data Validation. Retrieved 8 Sep, 2014 from https://www.owasp.org/index.php/Data_Validation#Data_Validation_Strategies

Robot Framework (2014). Retrieved 21 Oct, 2014 from http://robotframework.org

Saltzer, J.H. & Schroeder, M.D. (1975). The Protection of Information in Computer Systems. Retrieved Jul 30, 2014 from http://web.mit.edu/Saltzer/www/publications/protection/

SANS SEC504 (2014)

Wen Chinn Yew, wenchinn@outlook.com

Smith, R.E. (2012). Extract from A Contemporary Look at Saltzer and Schroeder's 1975 Design Principles. Security & Privacy, IEEE vol.10, no.6, pp.20-25, Nov-Dec 2012

Retrieved Jul 30, 2014 from http://cryptosmith.com/book/export/html/365

Sommerville, I. (2011). Software Engineering. Addison-Wesley

Tillmann, N. & Schulte, W. (2005). Parameterized Unit Tests. Microsoft Research. Retrieved 8 Sep, 2014 from http://research.microsoft.com/pubs/77419/ParameterizedUnitTests(FSE05).pdf

Viega, J. & McGraw, G. (2002). Building Secure Software: How to Avoid Security Problems the Right Way. Addison-Wesley

Wen Chinn Yew, wenchinn@outlook.com