Strategies for Secure Software Development

Sydney Pratte, Shena Fortozo, Gellert Kispal, Adesh Banvait and Alaa Azazi
Department of Computer Science
University of Calgary
Calgary, Canada
sapratte@ucalgary.com, shena.fortozo@gmail.com, kispalgellert@yahoo.com, adesh911@gmail.com, alaa.azazi@ucalgary.ca

Naomi Hiebert
Schulich School of Engineering
University of Calgary
Calgary, Canada
numberless1@gmail.com

In this paper we discuss the importance of assuring software security during development and with the use of testing and security products. We also discuss some of the common vulnerabilities in software that result in insecurities. These vulnerabilities include access control problems, insecure interaction between components, timing attacks, buffer overflows, and denial of service attacks.

Index terms - Software security, development, techniques, vulnerabilities, testing, tools

I. Introduction

In today's society, software is an omnipresent part of life. We trust it with everything from holding our money to helping fly airplanes, and for the most part, it deserves that trust. Software does a better job of keeping data safe yet instantly accessible than any conceivable non-computerized system could ever hope to achieve. However, as more and more valuable data is stored in computer systems, they become an ever more tempting target to those who could benefit from stealing or destroying that data. Businesses, for example, must store large amounts of sensitive and critical information on their systems, and it is of vital importance to keep this data away from prying eyes [11]. Businesses should also make software security a priority to assure reliability and integrity for both themselves and their clients.

Individuals, on the other hand, must be ever wary of attempts at identity theft. Identity theft is the act of stealing a victim's personal information such as bank account numbers, credit card numbers or social security numbers, and using that information to pose as the victim. If an attacker gains access to this information it can result in severe financial problems for the victim [12]. However, if proper software security measures are taken, personal information can be protected from these attackers.

Many different strategies have been integrated into software development methodologies to assure that security begins at development and continues to the final product. These strategies are employed through the entire product lifecycle, from initial requirements gathering all the way to deployment and maintenance. This paper will discuss two of these strategies: OWASP's CLASP and Microsoft's SDL. One of the ways these strategies recommend assessing the security of software is the use of software products that vigorously test for security holes and identify security vulnerabilities in your system. We will go into detail about two of these testing frameworks, namely the Metasploit project and the W3AF. In order for developers to take pre-emptive security measures during development and to be able to create secure software, they must understand software vulnerabilities. The 2011 CWE/SANS [1] list identifies 25 of the most dangerous software faults that can cause vulnerabilities. From this list, we discuss in detail five examples of software vulnerabilities.

II. During Development and in Products

Software security should be part of a project right from the onset and throughout the development. This section describes development methods, products, and testing methods that can be used to assure a secure product.

A. *Development Method*

The development process of secure software is still by large a matter of guidelines, strategies and personal expertise. Such strategies provide guidance in specific areas of software security such as threat modeling and testing.
During the recent years, researchers have developed a number of methodologies and techniques that employ the strategies and guidelines in these practices into integrated and comprehensive construction processes for secure software development.

The following section, briefly, discusses two of the forefront secure software development techniques, namely Microsoft's Security Development Lifecycle (SDL) [10] and OWASP's Comprehensive Lightweight Application Security Process (CLASP) [8]. It presents a high level introduction to both techniques alongside a description of the process phases of each technique.

1. Microsoft's Security Development Lifecycle (SDL)

SDL is a software security assurance process that is focused on software development [7]. Microsoft has adopted SDL as a mandatory policy in 2004 in order to resolve the security concerns that have previously arisen in its products [10].

The Microsoft SDL consists of a number of security activities presented in the order they must be implemented [9]. The activities are categorized into mandatory and optional activities, and are grouped by the phases of the software development life cycle (SDLC).

a. *Mandatory Security Activities*

In order to comply with the Microsoft SDL process, software must successfully pass the following six mandatory security activities.

i) *Pre-SDL Requirements: Security Training*

All team members who are directly involved with the development of the software must receive appropriate security training. The training must at a minimum cover the fundamental concepts of software security such as secure design, threat modeling, secure coding, security testing, and privacy.

ii) *Phase One: Requirements*

During the requirements phase, the development team performs security requirements gathering, risk assessment and establishes bug bars in order to plan the integration of security and privacy into the development process.

iii) *Phase Two: Design*

The Design phase identifies the design requirements and the structure of the software product. A core element of the SDL design phase is threat modeling which aids in the analysis of security issues in the internal components of the developed software.

iv) *Phase Three: Implementation*

In this phase, the development team decides and on and approves the set of tools that will be used during the development of the software such as compilers, linkers, libraries and APIs and assesses the security issues that could arise as a result of using these tools.

v) *Phase Four: Verification*

During the verification phase, the software is exposed to various types of dynamic tests in order to check against security and privacy specifications defined in the requirements and design phases.

vi) *Phase Five: Release*

During the release phase, the development team must create an incident response plan that identifies the roles of concerned personnel in case of an emergency. The team must also perform a Final Security Review (FSR) and archives all relevant data for future reference [7].

b. *Optional Security Activities*

Optional security activities are performed when the security of software is highly critical. These activities provide an additional level of certainty as well as in-depth security analysis for certain software components. This section provides a few examples of such activities.

i) *Manual Code Review*

Manual code review is mostly focused on the critical and the most sensitive components of the software and the security advisor or an expert in the field must perform it.

ii) *Penetration Testing*

Penetration testing aims to unveil potential security flaws and vulnerabilities through the simulation of attacks and the use of dynamic malformed random data.

iii) *Vulnerability Analysis of Similar Applications*

Investigating reputable vulnerabilities databases and similar software systems could aid in avoiding potential security issues during the design and implementation phases.

2. OWASP's Comprehensive Lightweight Application Security Process (CLASP)

CLASP is a lightweight software security assurance process that adopts an easy and effective approach for constructing secure software [8]. It aims at moving security concerns into the inception phase of the project by introducing extensions to the traditional software engineering activities and providing implementation guidance in certain security areas. The CLASP process consists of 24 top-level activities that can be fully or partially incorporated into software that is being constructed [8]. This paper categorizes and groups these activities according to their corresponding traditional software development life cycle (SDLC) phases.

a. *Education and Awareness Activities*

CLASP stresses that all team members must have adequate security training, and must be sufficiently familiar with the project's security policy. Some of the activities performed in this phase include providing an institute security awareness program, and appointing a project security officer.

b. *Requirements Gathering and Analysis Activities*

During the requirements phase, the team members must specify the operating requirements so that the impact on the security of the software can be evaluated. Some of the activities performed in this phase include identifying the project's global security policy, trust boundaries, user roles and detail misuse cases.

c. *Design Activities*

During this phase, the designers must apply the security principles that were agreed upon to the design of the software. Some of the activities performed in this phase include identifying attack surface, researching and assessing security posture of technology solutions, annotating class designs with security properties and specifying software-specific security configuration.

d. *Implementation Activities*

In this phase, the development team must integrate the security measures into the actual implementation of the software. Some the activities performed in this phase include integrating security analysis into source management process, implementing interface contracts and elaborating resource policies and security technologies.

e. *Testing and Verification Activities*

In this phase, the development team must assess the likely risks in the system and identify inadequate and improper security requirements. Some of the activities performed in this phase include threat modeling, addressing reported security issues, verifying security attributes of resources and performing source-level security review and tests.

f. *Deployment Activities*

In this phase, the development team provides a method for validating the integrity of the software, which is done through code signing.

g. *Update and Maintenance Activities*

After deployment, the team must continue to examine the software for potential security flaws and must use the resulting information to build a reference security guide.

B. *Development tools*

1. METASPLOIT PROJECT (FRAMEWORK)

Metasploit started as an open source (now owned by Rapid7) project that provides developers with the ability to identify security vulnerabilities using penetration testing. It is used to target vulnerable systems remotely using exploit code, and is considered one of the more popular frameworks in exploit development [15]. The framework is primarily used for finding vulnerabilities in software.

Recently, Metasploit helped researchers highlight vulnerabilities with Universal Plug & Play (UPnP), which affects millions of systems connected to the Internet. Researchers were able to find public IP addresses by scanning the targeted network and see if they got and response to UPnP requests. If they were able to find the IP address, it would give them an entry point into the network. Attackers can then choose to either attempt to find exploits in a specific device or, if the SOAP service is exposed, attempt to write an exploit to shift traffic and steal data from the user. This sort of penetration testing would allow IT teams to better fix these cracks and protect them from hackers [16].

Metasploit has a modular approach in building an exploit and allows combinations of any exploits with any payload. This gives more freedom to exploit designers. To write an exploit:

1. Choose and configure an exploit
2. Check whether the target system is susceptible to that kind of exploit
3. Choose and configure a payload
4. Encrypt the payload
5. Execute the exploit

Metasploit employs 'Fuzzing' techniques to achieve its goals. Fuzzing is an automated software testing technique that aims at finding Memory leaks, Assertion Failures and Exceptions. This is done using invalid and unexpected/random inputs for the program. We can think of fuzzing being employed in a fashion similar to black-box testing, the exploit writers have to set up a series of tests which can be executed when needed. It is a common technique used to find security problems in software and computer systems. There are two categories of fuzzing Programs:

1. Mutation Based – Mutate existing data samples by flipping bits or changing test suites and pass them as input streams to the program.
2. Generation Based – Define new test data based on models of the input.

The efficiency of fuzzing depends on the extent of code it can cover and the timeframe allotted for the test. Fuzzing is usually limited to finding simple bugs in the system, but helps designers in finding such bugs, which they might have overlooked or are unaware of.

2. W3AF – WEB APPLICATION ATTACK AND AUDIT FRAMEWORK

w3af is a open source vulnerability scanner and exploitation tool for web applications and web sites. It helps researchers identify vulnerabilities like SQL Injection, Cross-Site Scripting, guessable credentials, unhandled application errors and PHP misconfigurations. It employs a variety of tools including fuzzing testing.

w3af is divided into ‘Core’ and ‘plugins’ parts. Core coordinates the process with the plugins, and are configured and executed through a user interface [18]. There are over 130 plugins that are categorized into Discovery/Crawl, Audit, Grep, Attack, Output, Mangle, Evasion and Brute force.

A basic example of w3af is about SQL injections. Discovery/Crawl and Brute Force plugins are used to identify various forms and queries within the target web site and are followed by audit plugins that will employ fuzzing techniques to find SQL injections. Reports are generated for the user using output plugins.

III. Vulnerabilities

While the software industry is very prosperous business, it too has its drawbacks and dangers. There have been many publicized attacks and mistakes that developers made, which got exploited by hackers. Earlier software was seen as an unreliable tool intended for activities that weren’t considered serious or profitable. Activities such as blogging or gaming were considered to be insignificant. However, as developers started focusing on security and ensuring secure software, major corporations such as banks became confident and invested in the industry. Below are some examples of mistakes and vulnerabilities that software is subject to.

A. *Access Control Problems*

Access control is the authentication of who is allowed to do exactly what in your system. Many software security vulnerabilities arise from deficiencies or lack of access control. Some examples of access control problems are missing authentication, incorrect authentication and allowance of unlimited or numerous authentication attempts.

Providing no authentication can lead to major software security vulnerabilities because when there are no access control checks, users can access resources and perform actions that they should not be able to [1]. Software must provide authentication for functionality that requires user identity or accesses a significant amount of resources [1]. However, when designing a system, simply adding separate user privileges can protect this functionality. In the following Objective-C example the checkBankAccount() method does not check who is checking the bank account [1]:

```
@implementation Test

- (void) checkBankAccount: (NSString *)accountNumber {

    [account getBalance:accountNumber];
}

//...
```

In this simple example the user’s permissions should be authorized to ensure that the user checking this bank account object has the authority to view its balance. The following example shows how the code might be modified to provide authentication [1].

```
@implementation Test

- (void) checkBankAccount: (NSString *)accountNumber {

    if ( [user isAuthorized] ) {
        [account getBalance:accountNumber];
    }
}

//...
```

Part of the missing authentication access control problem is incorrect authentication. While missing authentication is more serious security vulnerability, incorrect authentication can also be an issue [1]. Incorrect authentication is when access control is applied to certain resources or actions but are implemented in a way that can be bypassed [1]. That is, when the resource or action that is accessed by the software does not correctly perform the authorization check. Both missing and incorrect authentication result in the same issue, users may be able to access resources or perform actions that they should not be able to do.

Another access control problem that results in software security vulnerabilities is the allowance of unlimited or numerous authentication attempts. Without a reasonable limit on the number of authentication attempts attackers can use brute force techniques to repeatedly guess different passwords until they succeed. Software is susceptible to this attack if it does not limit the amount of failed attempts in a short amount of time. A real world example of this issue is in 2009 an attacker who gained administrative access by taking advantage of this vulnerability accessed thirty-three celebrities and politicians Twitter accounts. The following example in Objective-C illustrates this vulnerability [1]

```
@implementation Test

- (BOOL) validateUser {

    BOOL isValidUser = NO;

    (NSString *) enteredUsername;
    (NSString *) enteredPassword;

    while ( !isValidUser )
    {
        enteredUsername = [self getEnteredUsername];
        enteredPassword = [self getEnteredPassword];

        if ( [enteredUsername length] > 6) {
            if ( [enteredPassword length] > 6)  {
                isValidUser = [self authenticateUser:enteredUsername
                    withPassword:enteredUsername];
            }
        }

    }
    return isValidUser;
}

//...
```

One possible solution for this example is to modify the authentication loop with a counter on the amount of attempts a user has performed. Once the counter reaches a reasonable maximum attempt value the system should handle the error [1]. The following code shows a method that checks a global count variable against a maximum attempt number:

```
@interface Test ()

@property (strong) NSString *count = 0;
@property (strong) int MAX_COUNT = 10;

@end


@implementation Test

- (BOOL) validateUser {

    BOOL isValidUser = NO;

    (NSString *) enteredUsername;
    (NSString *) enteredPassword;

    while ( !isValidUser && count < MAX_COUNT)
    {
        enteredUsername = [self getEnteredUsername];
        enteredPassword = [self getEnteredPassword];

        if ( [enteredUsername length] > 6) {
            if ( [enteredPassword length] > 6)  {
                isValidUser = [self authenticateUser:enteredUsername
                    withPassword:enteredUsername];
            }
        }
        count++;
    }
    return isValidUser;
}

//...
```

If the count exceeds the maximum, the method will return "NO" and the calling code must appropriately handle this situation. (Note that unlike in this example, many systems allow a small number of further attempts after a "cooldown" period have passed).

.

B. *Insecure Interaction Between Components*

This form of vulnerability arises from insecure ways of sending and receiving data from separate components, modules, programs, processes, threads, or systems. These mostly deal with user input and commonly attack database-driven applications. SQL injection and Cross-Site Scripting are two common examples of these types of weaknesses in software. According to the 2011 CWE/SANS [1] list, both are found on the top 25 most dangerous software errors that cause vulnerabilities.

1. SQL Injection

SQL injection happens when there is improper neutralization of elements that could alter an SQL statement. Those that have user-controllable inputs are particularly the ones that cause harm. Moreover, these changes can modify the query logic without being detected. An attacker can leverage this vulnerability by masking their SQL commands in a program's SQL code.

There are several consequences of successfully receiving an SQL injection [1]. One of the most common is the loss of confidentiality. Data could be changed or deleted all at once. Furthermore, this type of weakness could provide a gateway to steal or corrupt data. It may also create holes in the application's security by compromising access to system itself.

There are many variations of techniques to perform an SQL injection attack [1]. The most common mechanisms are injection through user input, cookies, and server variables. However, the use of second-order injection can be more difficult to detect and prevent, as the point of injection is different from the point where the attack actually manifests [4]. This is due to the fact that the data that may have past sanitation at one point may result in an attack when used in a different context or query. For example, the attacker could input ' OR 1 OR ' as their "username" when creating some sort of account. This doesn't necessarily cause a problem in the insertion query and therefore passes. This username is now stored in the database. However, in the context of another SQL query, which may be called later on, could alter the code logic due to the encapsulated OR logic symbol in the text field. In this case, the example would create two WHERE clauses when calling the attacker's username thus nullifying what comes next in the statement. This could alter the query logic entirely, weakening the integrity of the database.

The following example shows an SQL injection by dropping a table when the code logic was only to select certain fields depending on the user's email. The point of the attack was the user input "x'; DROP TABLE members; --", which is emphasized below:

```
SELECT   passwd, full_name
FROM     members
WHERE    email = 'x'; DROP TABLE members; --';
```

Prevention of an SQL attack can be done in both the architecture design and implementation phases [1]. The most evident, yet sometimes forgotten or done incorrectly, is the practice of defensive coding [4]. This entails cleaning and validating input and can be done by setting length limits on input fields, checking and setting data types, escaping special characters such as apostrophes and colons, and the use of alias and unique field names for information hiding.

There are at least two mitigation methods [2]. One is verifying that the database user should only have the minimum number of privileges required to run the application. This allows the amount of data that can be attacked to be isolated by limiting the user's access to other tables and/or commands. The other method is further encrypting the data stored. For example, passwords can be stored in a "salted hash" thus further securing the account information.

2. Cross-Site Scripting

On the other hand, Cross-Site Scripting (XSS) [1] is a vulnerability found in web applications that enables an attacker to embed malicious code into a legitimate web page. A user visiting this site is then fooled, and usually unaware, of executing the script on their machine.

Attackers may use this weakness by compromising private information, manipulate or steal cookies, or create a request on behalf of the victim [5]. This can be more dangerous if the victim has administrative access to a website. Additionally, the attacker could possibly take control over the victim's system, which is sometimes known as "drive-by hacking" [1]. Twitter and Facebook are examples of prominent sites that have been previously hit by this vulnerability [3].

There are various ways for an XSS attack to occur. One way is when the application does not sanitize or improperly sanitizes data from a web request or user input. Or, it could come from an external source such as dynamically and unknowingly generating a web page with malicious data [6]. In any case, XSS violates the web browser's "same-origin" policy [1], which says that a document or script from one origin should not interact with a resource from another origin.

There are three main types of XSS attacks, two of which has the injection being performed by the server. The first is called Non-Persistent or Reflected XSS [1]. This deals with the data sent through HTTP request being reflected back through an HTTP response. Typically, an XSS-tainted URL is sent to the user by means of email, publicly posted, etc. The victim would unknowingly be the supplier of dangerous content to the web application. The harmful script would be executed and reflected back to the user. The other type of common server-based XSS attack is called Persistent or Stored XSS [1]. This describes when injected code is stored on the target, usually on a database. Each request to view executes the malicious code. They usually occur in places that allow user input such as message boards and profiles. However, the third kind of XSS performs the injection on the victim's client. It is called DOM-based XSS or type-0 XSS [3]. This succeeds by modifying the victim's browser such that the page itself does not change. However, due to the modification, the page executes differently in the background.

There are various ways to help prevent an XSS attack. Proper escaping and quoting is the most effective solution. Another method is to perform security checks on both client and server side [1]. Specifying a proper character encoding scheme for every web page generated such that the web browser doesn't treat certain sequences as special when they are not implemented to be so. One other way is to use a whitelist of acceptable inputs as a strategy of validation.

SQL injection and Cross-Site Scripting are two of many ways to exploit vulnerabilities through the use of insecure interactions between components.

C. *Timing Attack*

A timing attack is a form of attempt to compromise a system by analyzing the time taken to execute a cryptographic algorithm [13]. Many complicated cryptographic algorithms are vulnerable to such attacks however in order to gain a good understanding of this vulnerability let's take a look at a simple code example below.

```
passwordCheck(String password, Account user) {

    if(password == user.password) {
        return true;
    }
    else
        return false;
}
```

By looking at this function no errors can be observed. It is a simple string comparison operation, which only grants a user access if all the characters in the input string match the characters in the user's password. However it is important to note the lower level details for this operation. Most compliers will do the following. Strings are represented as arrays of characters, so the if statement will iterate through both arrays and compare each character in the input string to the user's password. As soon as one discrepancy is found the if-statement will evaluate to false and the function returns. Consider input string "bcde" and password "abcd". With this input the function will return false right away because it will compare "a" to "b" and find that they're not equal. However if the input string is "abce" and password "abcd" the function will only return false when it iterates to the last character. From a time perspective, it will take longer for second input string to return false than the first. These comparison operations take very small amount of time (a few nanoseconds) and are not noticeable to humans;

however, all operating systems have a way of keeping track of time.

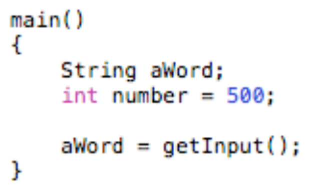
The attacker has the ability to brute force the password and as the response time increases it means that the input string closer resembles the user's password. Usually brute force algorithms that guess passwords have an exponential time complexity. However knowing that a system is vulnerable to timing attacks the time complexity can be reduced to linear time, because the attacker no longer has to guess every possible string, after each guess the attacker learns if they are getting closer to the password. Therefore such attacks can be executed within hours or even minutes.

Knowing that one's password can be guessed within minutes is a scary concept; however timing attacks only work in very specific conditions. First of all this specific attack cannot be executed over the network, meaning it is restricted to local systems. This is because the response time over a network is not conclusive. If one pings a server the response time for each ping is different due to network jitter. Since timing attacks reply on a very precise time measurement (to the millisecond) it is considered to be impossible to accomplish over networks. Timing attacks also assume that one has an unlimited number of attempts to try and guess the password.

There are many ways to prevent against this vulnerability. One could implement a cool-down rule where after a certain number of attempts the account locks. Another solution would be implementing a password check function that iterates to the end of the input string for every trial, therefore the attack is not given any clues whether they're getting closer to the password. That the input string closer resembles the user's password.

D. *Buffer Overflow*

Amongst the most common software vulnerabilities are buffer overflows. This is because C/C++ does not provide build-in protection against accessing memory outside of particular bounds. To grasp the idea of this concept below is a simple example.



```
main()
{
    String aWord;
    int number = 500;

    aWord = getInput();
}
```

Let aWord be an 8 byte array of characters and "number" a 16-bit integer (2 byte integer). To keep it simple let's assume that both variables are stored right next to each other in memory, so it is easy to visualize. The following diagram denotes how memory may look like using blocks to represent 1 byte of data [14].

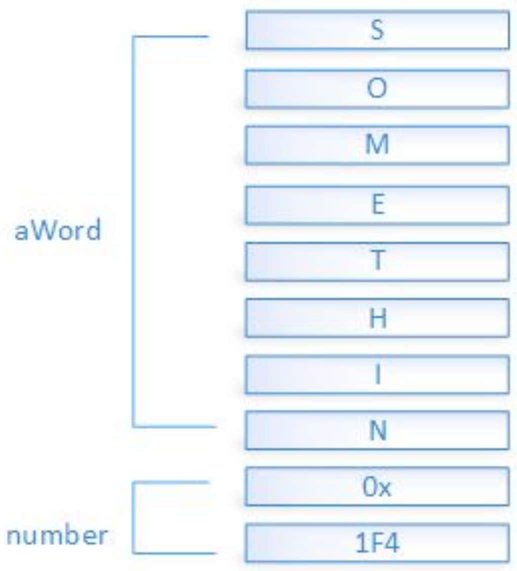


figure 1

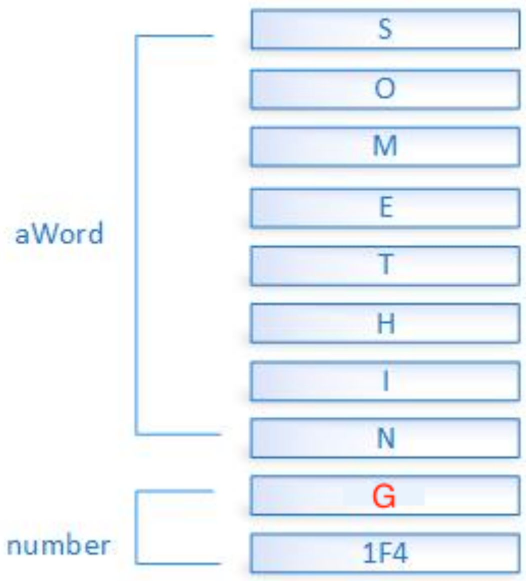


figure 2

Characters are encoded using ASII so the letter "S" would appear as a 73, however for simplicity let's ignore that momentarily. If the user inputs "SOMETHING" as string, it is greater than 8 bytes long, however the computer will put each character into their respective memory slots, demonstrated in figure 1. However when it reaches the end of the input it will result in overwriting the value in "number" as demonstrated in figure 2. Since our program is very small and we do not rely on the value of number, this exploit would make no difference. However let's say that "number" holds someone's account balance or other personal information like SIN number then this exploit would cause a major threat to people's privacy. Proper bounds checking would provide an easy fix for this vulnerability.

E. *Denial of Service*

While the above attacking methods were meant to penetrate systems and possibly destroy valuable data denial of service attacks are different in that perspective. As the name suggests this attack is only able to deny service to legitimate users, attackers do not gain access to the system. The basic flow of the attack [19]:

- the attacker makes a request to interact with the server (mostly likely from a spoofed IP address)
- the server replies and spawns a thread for interaction
- the attacker never replies, therefore the thread keeps hanging

If the attacker is able to send thousands of requests in a short period of time the server slows down considerably or even crashes entirely, since it is dealing with useless/fake interactions.

Denying service has no real value unless the particular service is important. For example if attackers are able to launch a Denial of Service on a bank website then the damage could result in millions of dollars due to the fact that many people heavily rely on system. Even a pizza delivering company can have major losses if hackers make their online orders inaccessible for legitimate users. In conclusion, while denial of service does not propose threat to user's privacy, it can do a fair amount to damage.

IV. CONCLUSION

Looking at software from a user perspective, one might not realize the complexity behind it. Security is a non-functional requirement, and tends to be largely transparent to users; however it can have a major effect on the end user. As software becomes more and more pervasive in our lives, more users will be exposed to the benefits and risks that come along with it. Engineers and designers must ensure that all software produced has a reasonable amount of attention paid to security. We must also avoid focusing too narrowly on security above all else, since security often comes at a cost not only in monetary terms but also in trade-offs with other important non-functional requirements.

Fortunately, there are some very well-understood ways to ensure that software is developed in a secure and reasonable manner. Use of security-aware development techniques such as SDL or CLASP allow security to be something built into the product at every stage of development. Tools such as Metasploit or W3AF allow security to be robustly tested, making developers aware of any potential faults or vulnerabilities. Finally, simply being aware of several of the more common attack vectors and how to avoid or confound them allows software developers to thwart attacks with a minimum of effort.

REFERENCES

[1] The MITRE Corporation, "(The MITRE Corporation, SANS Institute, 2011)" [online], Sept. 2011 [cited Feb. 22, 2013], available from World Wide Web: http://cwe.mitre.org/top25/

[2] Colin Angus Mackay, " SQL Injection Attacks and Some Tips on How to Prevent Them" [online], Code Project, Jan. 2005 [cited Feb. 22, 2013], available from World Wide Web: http://www.codeproject.com/Articles/9378/SQL-Injection-Attacks-and-Some-Tips-on-How-to-Prev

[3] Wikipedia, "Cross-site Scripting" [online], [cited Feb. 23, 2013], available from World Wide Web: http://en.wikipedia.org/wiki/Cross-site_scripting

[4] William G.J. Halfond, Jeremy Viegas, and Alessandro Orso, "A Classification of SQL Injection Attacks and Countermeasures" [online], [cited Feb. 23, 2013], available from World Wide Web: http://www.cc.gatech.edu/~orso/papers/halfond.viegas.orso.ISSSE06.pdf

[5] Acunetix, "What is Cross Site Scripting?" [online], [cited Feb. 23, 2013], available from World Wide Web: http://www.acunetix.com/websitesecurity/cross-site-scripting/

[6] Paul Lee, "Cross-site Scripting" [online], IBM, Sept. 2002 [cited Feb. 23, 2013], available from World Wide Web: http://www.ibm.com/developerworks/tivoli/library/s-csscript/

[7] Johan Gr´egoire, Koen Buyens, Bart De Win, Riccardo Scandariato, Wouter Joosen. DistriNet, Department of Computer Science, K.U.Leuven. "On the Secure Software Development Process: CLASP and SDL Compared", Celestijnenlaan 200A, B-3001 Leuven, Belgium

[8] OWASP Foundation, "OWASP CLASP v1.2 Comprehensive, Lightweight Application Security Process", OWASP. November 9, 2007 [cited February 22, 2013], available from World Wide

Web: https://www.owasp.org/index.php/Category:OWASP_CLASP_Project

[9] Noopur Davis, "Secure Software Development Life Cycle Processes: A Technology Scouting Report", Carnegie Mellon, Software Engineering Institute. December 2005.

[10] Microsoft Corporation, "Microsoft Security Development Lifecycle: Simplified Implementation of the Microsoft SDL" [online], November 4, 2010 [cited February 26, 2012], available from World Wide Web: http://www.microsoft.com/sdl

[11] Corr S. Pondent, "Importance of Security in Software Systems" [online], n.d. [cited Feb. 22, 2013], available from World Wide Web: http://www.ehow.com/facts_7528821_importance-security-software-systems.html

[12] Milton Kazmeyer, "Why is Security of Computer Systems Important?" [online], n.d. [cited Feb. 26, 2013], available from World Wide Web: http://www.ehow.com/facts_7528821_importance-security-software-systems.html

[13] Wikipedia, "Timing Attack" [online], [cited Feb. 26, 2013], available from World Wide Web: http://en.wikipedia.org/wiki/Timing_attack

[14] Wikipedia, "Buffer Overflow" [online], [cited Feb. 27, 2013], available from World Wide Web: http://en.wikipedia.org/wiki/Buffer_overflow

[15] Sectools.org, "Top 125 Network Security Tools" [online], December 7, 2012 [cited February 27, 2013], available from World Wide Web: http://sectools.org/tag/sploits/

[16] Tom Brewster, "How Attackers Can And Will Exploit UPnP Flaws" [online], January 30, 2013 [cited February 27, 2013], available from World Wide Web: http://www.techweekeurope.co.uk/news/how-attackers-will-exploit-upnp-105868

[17] Darren Pauli, "Search Phone Calls for Keywords with Metasploit" [online], February 6, 2013 [cited February 27, 2013] available from World Wide Web: http://www.scmagazine.com.au/News/331343,search-phone-calls-for-keywords-with-metasploit.aspx

[18] Sector, "A framework to 0wn the Web" [online], 2009 [cited February 27, 2013], available from World Wide Web: http://www.sector.ca/presentations09/w3af%20in%20150%20minutes%20-%20part%201.pdf

[19] Wikipedia, "Denial of Service Attack" [online], [cited Mar. 25, 2013], available from World Wide Web: http://en.wikipedia.org/wiki/Denial-of-service_attack