Universidade do Minho

Mestrado Integrado em Engenharia Informática

Engenharia de Segurança

Aula 7 - Vulnerabilidade de codificação

Grupo 8

João de Macedo
A76268

João Aloísio
A77953

Nelson Gonçalves
A78173

5 de Abril de 2020

1 Pergunta 1.1

1.1 1-

1.1.1 CWE-119

Em primeiro lugar com um *Score* de 75.56 temos CWE-119, em que o software executa operações em buffer de memória, mas pode ler ou escrever fora do limite desse buffer. Isto deve-se a que certas linguagens permitam o endereçamento direto sem que exista a verificação se esta memória se encontra ocupada ou não. Esta vulnerabilidade acontece mais em linguagens C e C++.

As consequências mais comuns afetam a nível de integridade, confidencialidade e disponibilidade quando um atacante executa um código não autorizado e consegue leitura e modificação de memória, que se efetivamente essa memória for controlada pelo mesmo pode redirecionar o ponteiro da função para o seu código malicioso, podendo assim ter acesso e alterar dados confidenciais e executar instruções que provoquem o *crash* do sistema.

1.1.2 CWE-79

Em segundo lugar temos CWE-79 com um *Score* de 45.69, que o software não neutraliza devidamente o input de um utilizador antes que chegue ao output de uma página web que seja utilizada por outros utilizadores. Esta vulnerabilidade prevalece mais nas tecnologia à base de web.

As consequências mais comuns afetam a nível de controlo de acesso, integridade, confidencialidade e disponibilidade, tendo em conta que o ataque mais comum é realizado com scripts entre web sites envolvendo a divulgação de informações armazenadas nas *cookies* do utilizador, ou seja, quando um utilizador mal-intencionado cria um script que permite realizar alguma atividade quando este é analisado pelo navegador, como por exemplo envio de *cookies* de um web site para um determinado email, onde pode conter informações confidencias de um utilizador.

1.1.3 CWE-20

Em terceiro lugar temos o CWE-20 com um *Score* de 43.61, em que o software não valida certos inputs que podem prejudicar o fluxo de controlo ou o fluxo de dados. Esta vulnerabilidade não prevalece em nenhuma linguagem específica.

As consequências mais comuns afetam a nível de integridade, confidencialidade e disponibilidade se um atacante conseguir um input malicioso que modifique os dados ou possivelmente conseguir alterar o fluxo de controlo de forma inesperada, assim como a execução de comandos arbitrários.

1.2 2-

A Weakness a ser tratada nesta pergunta será **Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')**, que tem o ranking número 11 (grupo 8 + 3 = 11).

Características:
O *software* constrói maior parte de um comando do Sistema Operativo usando *inputs* externos de uma componente anterior, no entanto, não impede totalmente que o comando seja modificado quando é enviado para a componente seguinte. Isto permite a que atacantes executem inesperados e perigosos comandos diretamente no Sistema Operativo. Se esta fraqueza ocorrer num programa com privilégios poderá dar ao atacante acesso a comandos com privilégios aos quais não deveria ter. Com isto o atacante pode comprometer gravemente o Sistema Operativo.

Linguagens:
Language-Independent

Consequências mais comuns:
Atacantes poderão executar comandos não autorizados, que posteriormente pode levar a comprometer a disponibilidade do *software*, ler ou modificar dados aos quais o atacante não tem permissões para aceder. A partir do momento em que a aplicação alvo está a executar diretamente os comandos ao invés do atacante, qualquer atividade maliciosa que ocorra parecerá ser proveniente da aplicação ou do seu dono, estando o atacante "escondido".

Com esta *Weakness*, os pilares da segurança de informação são comprometidos, sendo estes: **confidencialidade, integridade, disponibilidade e não-repudiação**.

Exemplo de código:
Este exemplo de código *PHP*, consiste em pegar no nome de um utilizador e obter a lista de conteúdo da *home directory* desse utilizador.

```
$userName = $_POST["user"];
$command = 'ls -l /home/' . $userName;
system($command);
```

A variável *$userName* não é verificada, portanto um atacante pode usar esta mesma variável para correr um comando arbitrário, tal como:

```
;rm -rf /
```

O que resultará no seguinte comando:

```
ls -l /home/;rm -rf /
```

Como o *';'* no sistema *Unix* é um separador de comandos, o resultado será a execução do comando *ls* e de seguida o comando que irá apagar todo o Sistema Operativo.

Exemplo de CVE: *CVE-2001-1246*:
Na *mail() function* é permitido, através dos argumentos, utilizadores locais e atacantes remotos executarem comandos via *shell*. Como não há neutralização do respetivo argumento, tanto como a *OS Command Injection (CWE-78)* e *Argument Injection (CWE-88)*, são possíveis.

2 Pergunta 1.2

2.1

Tendo em conta que usualmente se utiliza a estimativa de que um software desenvolvido com métodos rigorosos de segurança contém entre 5 a 50 bugs por 1000 linhas de código fonte e software desenvolvido, podemos realizar as seguintes estimativas:

- **Facebook** tem cerca de 62 milhões de linhas de código, portanto, pelo o seu código poderá ter entre 310 mil a 3 milhões de bugs.
- **Software de automóveis:** em média um software de automóveis tem 100 milhões de linhas de código, logo, este software pode conter entre 500 mil a 5 milhões de bugs no respetivo código.
- **Linux 3.1:** uma vez que o Linux 3.1 possui cerca de 15 milhões de linhas de código, pode-se concluir que este sistema tem entre 75 mil a 750 mil bugs no seu código.
- **Todos os serviços Internet da Google** tem cerca de 2 mil milhões, pelo que tem entre 10 milhões a 100 milhões de bugs.

2.2

É praticamente impossível dizer quantas vulnerabilidades existem nos bugs estimados, visto que num número muito grande de bugs podem existir muito poucas vulnerabilidades e num número muito pequeno de bugs podem existir muitas vulnerabilidades, dependerá sempre do tipo de bug que é.

3 Pergunta 1.3

Para as vulnerabilidades de projeto:

- **Insufficient UI Warning of Dangerous Operations :** acontece quando, na fase de design, se ignoram situações em que, avisos de operações perigosas devem ser postos, de uma forma clara, à atenção do utilizador, o que pode resultar numa quebra de segurança.

- **Improperly Implemented Security Check For Standard :** acontece quando, na fase de design, não se tem em conta que um software não está a implementar completamente um algoritmo/técnica standardizada, o que pode resultar numa quebra de segurança, pois existe a possibilidade desse algoritmo ou técnica ser vulnerável.

Para as vulnerabilidades de codificação :

- **Improper Input Validation :** acontece quando um programador nega o facto de que um atacante conseguir modificar o input de campos escondidos ou de cookies, quando recorre a proxies, por exemplo.

- **Path traversal:'../filedir : acontece quando o software usa input externo para construir um nome de caminho para um ficheiro que devia estar dentro de uma diretoria restrita, mas que não neutraliza sequências de '../' que podem direccionar para diretorias fora da diretoria restrita.**

Para as vulnerabilidades operacionais :

- **ASP.NET Misconfiguration: Password in Configuration File :** acontece quando, ao configurar um software, se guarda uma password sem ser cifrada, permitindo que qualquer pessoa que ganhe acesso a esse ficheiro consiga ler essa password.

- **ASP.NET Misconfiguration: Creating Debug Binary :** acontece quando, após a fase de desenvolvimento e testes, são mantidas mensagens de debug detalhadas, o que é um risco de segurança para o software.

4 Pergunta 1.4

Uma vulnerabilidade dia-zero é uma vulnerabilidade de codificação que ainda não foi tornada pública a toda a comunidade informática, apenas um grupo restrito tem conhecimento da mesma. Pelo contrário, as vulnerabilidades de codificação "normais"são do conhecimento geral de toda a comunidade informática.