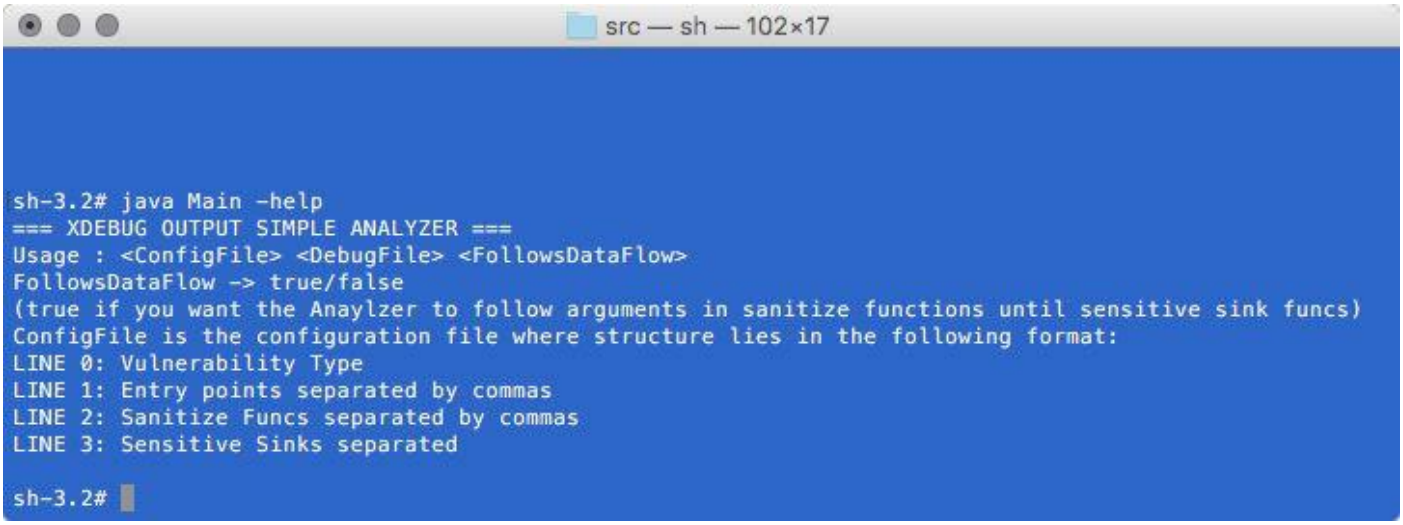
TÉCNICO LISBOA

MEIC-A: **Segurança em Software**

Relatório – Tema A – Deteção de Ataques contra aplicações PHP



Elaborado por:

João Aguiar – 76471
Carlos Correia – 76512
Filipe Custódio – 84938

1. Ferramenta Experimental
    a. Abstrato:

O uso da ferramenta desenvolvido centra-se na análise dos outputs produzidos por um debugger de php amplamente usado e com um vasto suporte. Tais outputs são na maioria das vezes difíceis de analisar, até para o seu utilizador central, o programador. Usualmente, estes são tratados, através de *parsing* por IDEs de PHP conhecidos, i.e *PHPStorm*, resultando na ativação de uma extensão de *Debugging*, fácil de usar, com acesso a valores de variáveis, tempos de execução e tantos outros.

Para a nossa ferramenta, queremos pegar nesses outputs e extrair funções específicas usadas, tal como o seu fluxo de execução. São estas: funções vulneráveis a ataques, nomeadas de *sensitive sinks*, e funções que protegem ataques, normalmente chamadas à priori das anteriores, denominadas de *sanitize functions*. Queremos também configurar a ferramenta com um número de padrões que contém, cada um, o nome da venerabilidade em questão, as funções vulneráveis e as *sanitize functions* a usar para as proteger. A partir da lista de funções extraídas, tal como o fluxo de variáveis entre elas, cruzado com os padrões carregados, a ferramenta lista com detalhe as secções do código que poderão estar desprotegidas, e as que, em principio estão protegidas. No caso das desprotegidas, é sugerido um leque de funções a utilizar para resolver o problema específico.

    b. Configuração do Debugger (*XDebug*):

No procedimento experimental efetuado, foi usada a extensão PHP Xdebug versão 2.4.0 rc1, a qual foi configurada no php.ini com os seguintes parâmetros:

```
xdebug.collect_params = 4
xdebug.trace_format = 1
xdebug.collect_vars = 1
xdebug.collect_return = 1
```

Figura 1- Configuração do XDebug no php.ini.

Esta configuração vai definir o formato do output a tomar, feito o trace. Começámos por definir o parâmetro *xdebug.collect_params* a 4 para permitir que o *XDebug* colecione os nomes e valores dos parâmetros das funções, quando estas são chamadas. De seguida, optámos por escolher o *xdebug.trace_format* a 1, o que fez com que o output gerado seja apresentado num formato computorizado (*computer readable*), o que, apesar de dificultar a análise por humanos, permite uma mais fácil interpretação pela nossa ferramenta, pois o formato está parametrizado. Apesar de o *xdebug.collect_vars* não gerar diferença no output do trace, optámos também por o definir como *enable*, pois assim poderíamos colecionar todas as variáveis declaradas, através do comando *xdebug_get_declared_vars()* no ficheiro php. Por fim, com o objetivo de analisarmos o retorno de cada função, tivemos de activar a opção *xdebug.collect_return,* permitindo um melhoramento da nossa ferramenta através da análise de todo o "*dataflow*".Após a configuração, deu-se um trabalho de pesquisa de ficheiros php passivos de possíveis vulnerabilidades, nos quais executámos alguns traces. Deixamos aqui, o ficheiro php que selecionámos para explicar o funcionamento da nossa ferramenta:

```php
<?php
xdebug_start_trace();
require_once("config.php"); // configuracao bd mysql

$id=$_GET['id'];
$escaped_id = mysql_escape_string($id);

$resulta = mysql_query("select * from churrascos WHERE churrasco_id = ".$escaped_id);
$resultb = mysql_query("select * from churrascos WHERE churrasco_id = ".$escaped_id);

// Do domething with the results

xdebug_stop_trace();
?>
```

Figura 2- Ficheiro php passivo de vulnerabilidades.

Para dar inicio ao trace do *XDebug*, temos de inserir os comandos *xdebug_start_trace()* e *xdebug_stop_trace()* nos limites do código executado. Após esta edição, acedemos ao ficheiro php através do browser e é automaticamente gerado o seu trace para a pasta do *XDebug* (neste caso, acedemos com ao *service_churrascos.php?id=30*).

Apresentamos de seguida uma síntese do output gerado pelo debugger:

```
Version: 2.4.0rc1
File format: 4
TRACE START [2015-12-04 19:50:17]
2    1    1    0.015755    130864
2    2    0    0.084266    132776  require_once    1    C:\xampp\htdocs\config.php  C:\xampp\htdocs\service_churrascos.php  3    0
// Connect to the DB and load the config
2    5    0    0.129718    138160  mysql_escape_string 0       C:\xampp\htdocs\service_churrascos.php  6    1    '30'
2    5    1    0.130129    138160
2    5    R           '30'
2    6    0    0.130220    138344  mysql_query 0       C:\xampp\htdocs\service_churrascos.php  8    1    'select * from churrascos WHERE churrasco_id = 30'
2    6    1    0.198039    140712
2    6    R           resource(5) of type (mysql result)
2    7    0    0.198108    140800  mysql_query 0       C:\xampp\htdocs\service_churrascos.php  9    1    'select * from churrascos WHERE churrasco_id = 30'
2    7    1    0.199299    142736
2    7    R           resource(6) of type (mysql result)
2    8    0    0.199351    142728  xdebug_stop_trace   0       C:\xampp\htdocs\service_churrascos.php  13   0
                  0.199395    142776
TRACE END   [2015-12-04 19:50:17]
```

Figura 3- Síntese do trace gerado pela execução do php apresentado.

Dado este output, a ferramenta foi executada duas vezes, com diferentes parâmetros:

```
ALERT: There is probably no vulnerability!
--> File: C:\xampp\htdocs\service_churrascos.php Line: 8
--> Function mysql_query
--> Arguments: select * from churrascos WHERE churrasco_id = 30
Sanitize Function used: mysql_escape_string
--> File: C:\xampp\htdocs\service_churrascos.php Line: 6
--> Sanitized args: 30

ALERT: There is probably no vulnerability!
--> File: C:\xampp\htdocs\service_churrascos.php Line: 9
--> Function mysql_query
--> Arguments: select * from churrascos WHERE churrasco_id = 30
Sanitize Function used: mysql_escape_string
--> File: C:\xampp\htdocs\service_churrascos.php Line: 6
--> Sanitized args: 30


WARNING: There is probably a vulnerability!
--> File: C:\xampp\htdocs\service_churrascos.php Line: 3
--> Function require_once
--> Arguments: C:\xampp\htdocs\config.php
SUGGESTION: Use one of these sanitize functions:
--> san_mix
```

Figura 4 –exec com "Java Main config.txt trace.xt true"

```
ALERT: There is probably no vulnerability!
--> File: C:\xampp\htdocs\service_churrascos.php Line: 8
--> Function mysql_query
--> Arguments: select * from churrascos WHERE churrasco_id = 30
Sanitize Function used: mysql_escape_string
--> File: C:\xampp\htdocs\service_churrascos.php Line: 6

WARNING: There is probably a vulnerability!
--> File: C:\xampp\htdocs\service_churrascos.php Line: 9
--> Function mysql_query
--> Arguments: select * from churrascos WHERE churrasco_id = 30
SUGGESTION: Use one of these sanitize functions:
--> mysql_escape_string mysql_real_escape_string

WARNING: There is probably a vulnerability!
--> File: C:\xampp\htdocs\service_churrascos.php Line: 3
--> Function require_once
--> Arguments: C:\xampp\htdocs\config.php
SUGGESTION: Use one of these sanitize functions:
--> san_mix


@@ END VULNERABILITES INFO @@
```

Figura 5 –exec com "Java Main config.txt trace.xt false"

O terceiro parâmetro da execução da aplicação (em JVM) define se a política de análise considera o fluxo das variáveis entre o retorno das funções de sanitização e funções sensíveis. Olhando para a figura da esquerda, onde este fluxo é considerado, temos um resultado coerente, ao passo que na figura à direita o programa apenas analisa o nome das funções no decorrer da execução, originando um falso positivo. Isto porque a ferramenta apenas segue o fluxo de execução sem analisar fluxo de dados e o conteúdo dos mesmos.

Estas duas abordagens de análise foram desenvolvidas, dado que em certos casos, seguir o fluxo de dados pode originar falsos negativos, dada a complexidade e ofuscação que poderá estar associada ao código a testar.

c. Disclaimer:

O formato de output do debugger mais preciso e bem definido que foi configurado, de forma a facilitar a recolha de informação pela ferramenta, contém diversas limitações que colidem com as especificações do projeto. A mais notória é a impossibilidade de recolher "*entry points*". Existem outras funções descartadas pelo *debugger*, nomeadamente o *echo, print, eval e die*, que no caso de um dos padrões para XSS, tem alguma notoriedade. Uma possível solução para a recolha dos "*entry points*" seria o uso de uma análise estática do código PHP, o que em suma originaria uma análise hibrida.

A lista de padrões criada é baseada no trabalho da professora Ibéria Medeiros, o WAP.

2. State of the Art

Nos últimos anos as aplicações web têm evoluído de simples páginas estáticas para estruturas dinâmicas muito bem pensadas. Tal evolução tem custos que são facilmente identificados ao utilizar uma enorme quantidade de linhas de código e métodos de linguagens de programação diferentes numa só aplicação. Para complicar ainda mais esta realidade, as aplicações web têm de aceitar e processar centenas ou mesmo milhares de parâmetros introduzidos pelos utilizadores e têm de continuar a ser sistemas seguros. Um exemplo de toda a complexidade envolvida é o artigo de Patrik Fehrenbach (*Discovering SQL Injection Vulnerabilities*) na revista ADMIN, Network & Security, em que este refere que os ataques de injeção de código SQL já existem há 12 anos e mesmo com todos os avanços continuam a ser uma das mais populares formas de ataque utilizada pelos hackers. *SQL Injection* é apenas uma vulnerabilidade de uma linguagem existente entre as múltiplas utilizadas numa aplicação web, o que amplia a motivação para os inúmeros estudos teóricos sobre como combater todos os vetores de ataque, sendo que ainda não existe uma solução generalizada.

Em segurança informática para a deteção de vulnerabilidades existentes no código estão generalizados dois tipos de abordagens. Uma abordagem estática (*static analysis*), que muito resumidamente envolve uma análise integral do código, através do estudo de todos os resultados e caminhos originados pelos diferentes parâmetros introduzidos pelo utilizador. Uma abordagem dinâmica (*dynamic analysis*), que se baseia em seguir os dados aceites à medida que estes percorrem o código fonte da aplicação e vai analisando a sua utilização de modo a identificar problemas de segurança. No âmbito das aplicações web, dada a quantidade de combinações de linguagens e caminhos, tal como o número astronómico de possibilidades dos dados introduzidos (levando à geração de muitos avisos falsos positivos – não sendo aceitável), só faz sentido a utilização de uma abordagem dinâmica (ou no máximo híbrida).

Dentro deste contexto existem algumas estratégias que identificamos como mais relevantes e utilizadas na descoberta de vulnerabilidades, sendo estas a *taint analysis* e a *forward symbolic execution*.

A estratégia de *taint analysis*, marca todos os dados recebidos de uma fonte exterior não confiável (ex: dados introduzidos pelo utilizador) e segue essas marcas até serem identificadas vulnerabilidades ou serem transformadas em dados confiáveis. Uma vulnerabilidade é originada quando um destes objetos marcados é mal utilizado pela aplicação, como por exemplo, ao ser diretamente introduzido numa *query SQL* ou apresentado ao(s) utilizador(es). Os dados marcados podem passar de não confiáveis para confiáveis quando tratados por funções que o asseguram (*sanitization functions*). Esta estratégia pode ser aperfeiçoada para seguir pormenorizadamente todo e qualquer input não confiável (*precise tainting*), ao ponto de se conseguir destingir, após uma concatenação de *strings* ou alguma função que altere *strings*, quais os caracteres que podem ser prejudiciais para a aplicação e quais aqueles que foram originados pela aplicação e nunca influenciados por qualquer dado não confiável.

A estratégia *forward symbolic execution* baseia-se em representar por símbolos as variáveis do sistema de modo a criar facilmente restrições para os caminhos possíveis. Estas restrições, determinam antecipadamente que *inputs* é que causam o quê e consequentemente vulnerabilidades existentes na aplicação. Para além destas estratégias, é também de salientar a abordagem de Davide Balzarotti, Marco Cova, Vika Felmetsger, Nenad Jovanovic, Engin Kirda, Christopher Kruege e Giovanni Vigna, que ao invés de ir à procuram de vulnerabilidades tiveram a ideia de verificar se os dados não confiáveis eram realmente confiáveis após tratados por *sanilitization functions*. Isto foi provado em muitos casos não ser o sucedido (no paper Saner*: Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications*). Resumidamente, nesta interessante abordagem, os autores criam uma estratégia hibrida que utiliza autómatos finitos para representar caracteres de *strings* e a propriedade de confiança. Usam também grafos de dependência para seguir os dados nos diferentes momentos da aplicação. Esta estratégia hibrida é composta por uma análise mais estática em que se procura os diferentes caminhos possíveis por *input* não confiável (relacionada com *taint analysis*) e se encontram ou não *sanitization functions* e por uma análise dinâmica aplicada a todas as situações identificadas como problemáticas na análise estática (reduzindo substancialmente), que utiliza os grafos de fluxo para descobrir ligações a processos de sanitização e verificar se estes realmente originam dados confiáveis.

Todas as estratégias para identificação de vulnerabilidades aqui descritas e investigadas dão-nos fortes indícios de melhorias numa aplicação web em vários vetores de ataque (*XSS, SQL Injection, PHP Injection*, entre outros...).

Em termos de performance, estas estratégias implicam o gasto de alguns recursos, mas nada demasiado significativo que possa ser considerado mais importante que o combate a vulnerabilidades. Os resultados obtidos por tais implementações, como já referido, vão usufruir de diminuir em muito a quantidade de deteção falsas positivas versus análise estática, sendo portanto a escolha mais acertada para os programadores. Mesmo assim, apesar do aperfeiçoamento destas implementações não podemos assegurar que todas as vulnerabilidades serão encontradas.

Tendo em consideração estes algoritmos de análise, seria possível melhorar a nossa ferramenta utilizando, por exemplo, "*Dynamic Taint Analysis"*. Para aplicar este tipo de análise, teríamos de adaptar a nossa ferramenta para fornecer um maior poder de *tracing*. Isto poderia ser obtido através de uma memória dinâmica para marcar variáveis como confiáveis ou não confiáveis. Durante o decorrer da análise, o valor de confiança destas seria verificado, originando ou não detecção de vulnerabilidades.

Em alternativa, podemos ainda melhorar a ferramenta através da abordagem *Saner*, verificando a eficácia da sanitização. Sempre que encrontrássemos uma função de sanitização teríamos de verificar se esta era ou não a adequada para a respetiva vulnerabilidade a ser tratada. Este processo poderia ser realizado através da aplicação de uma análise dinâmica, precedida de uma análise estática, como já mencionámos anteriormente na atividade experimental. Esta abordagem poderá requerer a criação de uma nova linguagem ou o recurso a uma linguagem com um elevado poder de recursão (ex: Haskell).