Escola de Engenharia
Universidade do Minho

DEPARTAMENTO DE ENGENHARIA INFORMÁTICA
Mestrado em Engenharia Informática
Engenharia de Segurança

Ferramentas e técnicas de *Compiler Warnings*

Grupo 1 e Grupo 3

Adriana Meireles
A82582

Carla Cruz
A80564

Ricardo Pereira
A73577

Tiago Ramires
PG41101

Braga, 11 de Maio de 2020

1. Introdução

A área da engenharia de segurança consegue ser extremamente vasta, abrangendo inúmeras áreas de outros campos da ciência. Nos dias que correm é quase impossível não estar perto de um dispositivo que necessite de um determinado tipo de proteção, seja um sensor biométrico, um *smartphone*, um router, entre outros. Se atentarmos a outros casos, apercebemo-nos que a engenharia de segurança é também aplicada em dispositivos médicos, dispositivos bancários, sendo por isso uma área extremamente requisitada para conceber e assegurar a proteção dos mais diversos sistemas que podemos encontrar no nosso dia a dia.

Contudo, apesar de existir um ramo da engenharia orientado à segurança de sistemas, é muito frequente vermos a ocorrência de falhas extremamente básicas, muitas das vezes detetadas após ataques bem sucedidos, que implicam custos extremamente avultados para as organizações que deles foram alvo. Assim, a principal questão impõe-se: prevenir ou remediar?

No projeto em questão é feita uma investigação aprofundada dos compiladores, explorando tudo aquilo que os mesmos podem oferecer para além da compilação dos programas, que é a principal função para a qual são desenhados. Mas será que se sabem as outras tantas utilidades que um compilador pode ter? Será que os programadores dos dias de hoje estão cientes dos seus benefícios no código que produzem? A ocorrência de *warnings* tem vários propósitos, entre eles reduzir as vulnerabilidades que se verificam nos *softwares* criados e por isso devem ser tidos em conta em paralelo com outras ferramentas de verificação de *software*.

Crê-se que a grande maioria das organizações já saiba o que é remediar, pois os ataques são frequentes e muitas vezes não necessitam de grande conhecimento da área para serem perpetuados, ou seja, atualmente os problemas de segurança surgem já após a implementação da aplicação estar feita e a sua resolução passa por colocar um ou mais técnicos informáticos a mitigar o problema. Já a prevenção, ainda é algo estranho às organizações, principalmente às empresariais, visto que existe sempre muito trabalho a ser feito, e fazer um esboço das medidas de segurança a implementar parece sempre ser uma perda de tempo. É este tipo de problemas que são abordados neste relatório, bem como as soluções para os resolver. A divulgação da área e das ferramentas existentes ou que poderão vir a existir são também aspetos importantes a salientar.

2. Desenvolvimento

2.1 Compiladores GCC

2.1.1 Funcionamento

Como é do conhecimento da maioria dos programadores, a compilação de programas em linguagens do tipo *C* é diferente da compilação de programas escritos noutras linguagens. A forma de compilação mais básica pode ser feita com recurso ao comando *"gcc <nome do ficheiro com o código do programa>"* e é a este mesmo comando que se podem acrescentar algumas opções do comportamento do compilador que se queiram ver incluidas na compilação. O resultado do comando anterior seria um ficheiro executável denominado *"a.out"*, mas caso existisse a necessidade de dar um nome específico ao executável resultante, podia incluir-se a opção "*-o <nome do executável>*". Da mesma forma que existe esta opção, existem outras tantas que um programador pode usar para trabalhar com o seu programa, nomeadamente, opções que lhe permitem verificar a qualidade e consistência do seu próprio código de uma forma automática e simples.

Ora, sabemos que a produção de código em linguagem *C* é provavelmente umas das mais complexas formas de programar que existem, uma vez que é possível interagir diretamente com as posições de memória e com o conteúdo das mesmas, algo que acrescenta alguma perigosidade aos programas. Posto isto, o *GCC (GNU Compiler Collection)* oferece uma panóplia de ferramentas que permitem a deteção de possíveis problemas que possam ocorrer durante a execução do programa, sendo os mesmos apresentados ao programador após a conclusão/tentativa da/de compilação. Aqueles em que o relatório em questão se foca são os ***warnings***, que normalmente aparecem quando o programador não segue as boas práticas que são intrínsecas a este tipo de linguagens. A seguir enunciamos alguns tipos de ***warnings*** que podem aparecer posteriormente à compilação de um programa.

2.1.2 *GCC* e *G++*

É sabido, da documentação oficial disponibilizada, que o *GCC* é um compilador que foi adptado para várias linguagens de programação, todas relacionadas com *C*. Essencialmente, o compilador é constituido pela parte *backend*, que é comum a todas as linguagens, e pela parte *frontend*, que é específica para cada linguagem que é aceite pelo *GCC*. Posto isto, é possível compilar código em *C++* com o *GCC*, contudo, a compilação de programas escritos nesta linguagem faz-se sobretudo usando *G++*, que é a componente do *GCC* que está totalmente afeta a programas desse tipo.

Em termos de opções de *warnings*, existem alguns que são comuns a qualquer linguagem e outros que são específicos. A título de exemplo, a opção "*-Wdeprecated-copy*"está disponível apenas para *C++*, enquanto que a "*-Wsign-compare*"aplica-se somente a *C*. Tendo em conta

que estas diferenças são mínimas, analisar-se-á tudo, doravante, na prespetiva da linguagem *C*.

2.1.3 Diferenças entre *errors* e *warnings*

Durante a compilação de um programa em linguagem *C* podem ocorrer *errors* e/ou *warnings*, ou na melhor das hipóteses não ocorre nenhum dos dois, um bom indicador de que o programa está a seguir boas práticas. Contudo existe uma grande diferença entre ambos os problemas: os *errors* impossibilitam a obtenção de um executável e por isso, impossibilitam também a execução do programa, já os *warnings* apenas advertem das consequências da má implementação que está a ser feita, criando-se posteriormente o ficheiro de *output* que pode ser executado. Normalmente, vêm-se *errors* quando existem inconsistências na tipagem de variáveis, declaração e construção de ciclos, má aplicação de valores de retorno de funções, entre outras. Por sua vez, os *warnings* anunciam, por exemplo, a manipulação de variáveis que estejam relacionadas mas que não são do mesmo tipo, o uso de funções que não dão garantias do bom funcionamento do programa, a declaração de variáveis que não estão a ser utilizadas, etc.

Porém, existem opções de compilação que uma vez ativadas, levam a que problemas que normalmente seriam meros *warnings* passem a ser *errors*, precisamente para forçar e obrigar o programador a resolver o problema.

2.1.4 *Segmentation faults*

A ocorrência de *segmentation faults* é talvez o tipo de *errors* mais difícil de solucionar da programação em linguagem *C*, pelas mais variadas razões. Mas a que é que se devem na realidade estes *segmentation faults*? Um *segmentation fault* é um erro do programa que ocorre durante a execução do mesmo, motivado pelo acesso indevido a espaços da memória que não estão alocados à tarefa que está a ser processada e pelo facto de só ser possível detetá-los durante este período, não aparecem como *errors* aquando da compilação. Muitas vezes, estes problemas não chegam sequer a ser detetados porque a única forma de o serem é através de comportamentos muito específicos quem nem sempre ou raramente acontecem durante a execução.

Uma das inúmeras funções dos *warnings*, é identificar práticas perigosas que possam levar o programa a situações de *segmentation fault*. Por exemplo, o uso da função *gets()* da *library stdio.h*, origina muitas vezes um *warning* que alerta que a mesma é "perigosa e não deve ser usada". O problema será explicado em detalhe mais à frente, mas este é um exemplo de um *warning* que pode prevenir com eficácia erros deste tipo.

```c
#include <stdio.h>
extern int printfile(FILE *pointer);
extern int wc(FILE *pointer);

int main(int argc, char **argv)
{
        FILE *pointer;

        if( argc > 1 )
                pointer = fopen(argv[1], "r");
        else
                pointer = stdin;

        printfile(pointer);

        pointer = fopen(argv[1], "r");

        wc(pointer);

        fclose(pointer);
        return 0;
}
```

Figura 2.1: Programa propício à ocorrência de *segmentation fault*.

Outra característica muito importante dos *segmentation faults* é o facto destes poderem representar vulnerabilidades para o próprio programa ou até mesmo para a máquina que o corre. Caso a vulnerabilidade seja explorada, pode traduzir-se num ataque e aí pode adquirir dimensões inimagináveis, como aconteceu no caso *heartbleed*, onde os *hackers* podiam ter acesso às chaves privadas das máquinas que invadiam, tirando proveito de *read overflows*.

Uma forma de compreender eficazmente a origem dos *segmentation faults*, passa pela utilização do *GDB* [10], uma ferramenta *GNU* que possibilita a interação do programador com o programa durante a execução do mesmo. Para o utilizar é necessário incluir a opção de compilação que indica ao compilador que deve incluir informação adicional no executável a ser utilizado pelo *GDB*. Com esta ferramenta, é possível "olhar para dentro"dos estados da máquina tal como eles existem e verificar se de facto têm o que era esperado.

2.1.5 Vulnerabilidades

Já foi possível compreender que os conceitos vulnerabilidade, *compiler warning* e *segmentation fault* estão bastante interligados. Os *compiler warnings* previnem a ocorrência de possíveis *segmentation faults* que podem resultar em vulnerabilidades para o programa. O facto de ser algo recorrente permitiu aos especialistas nomear e separar as "espécies"de vulnerabilidades existentes, sendo algumas enunciadas a seguir:

- ***stack buffer overflow*** - é uma vulnerabilidade que está relacionada com a *stack*, onde são declaradas as variáveis das funções e a informação para as chamadas das mesmas, que permite, por exemplo, executar funções que não deviam ser executadas;
- ***heap buffer overflow*** - é uma vulnerabilidade que está relacionada com a *heap* onde são feitas todas as alocações dinâmicas de memória - permite a leitura de variáveis que não são mostradas ao utilizador;
- ***read overflow*** - caso particular de uma vulnerabilidade de *buffer overflow* onde é possível fazer a leitura de dados da memória.
- ***integer overflow*** - vulnerabilidade que acontece quando o limite da variável desse mesmo tipo é ultrapassado, levando a que seja escrito um valor completamente diferente do que era suposto.

2.1.6 *Warnings*

Como foi referido anteriormente existem várias opções para ativar/desativar a ocorrência de determinados tipos de *warnings* e em baixo mencionamos alguns[9]:

- "***-w***- impede que sejam apresentados *warnings* após a compilação;
- "***-Werror***- faz o compilador interpretar os *warnings* como se de erros se tratassem;
- "***-Werror=<tipo de warning>"*** - tem a mesma função que a opção anterior mas para um determinado tipo de warnings;
- "***-Wfatal-errors"*** - embora relacionada com *errors*, esta opção obriga o programador a corrigir o primeiro erro que o compilador deteta, não apresentando mais nenhum;
- "***-Wpedantic"*** *e "****-pedantic"*** - apresenta *warnings* quando os padrões *ISO C* e *ISO C++* não são cumpridos;
- "***-Wall***- ativa a emissão de uma grande panóplia de *warnings* que têm que ver com a contrução de código e de boas práticas de programação;
- "***-Wextra***- complementa a opção anterior com a ativação de mais alguns tipos de *warnings*.

Seria possível enumerar mais alguns, contudo a lista ainda é bastante extensa e lá podemos encontrar *warnings* que avisam, por exemplo, da má construção de um *switch case*.

```
switch (cond)
  {
  case 1:
    a = 1;
    break;
  case 2:
    a = 2;
  case 3:
    a = 3;
    break;
  }
```

Figura 2.2: *Switch case*.

Posto isto, na subsecção seguinte apresentar-se-ão compilações de programas sem e com algumas opções apresentadas anteriormente.

2.1.7 *Warnings* na prática

Para se demonstar a ocorrência de *warnings* e verificar se existe ligação com alguma vulnerabilidade de *buffer overflow* compilaram-se os programas localizados num repositório do *github*. Todos eles podem ser acedidos através deste *url* - `https://github.com/uminho-miei-engseEngSeg/tree/master/TPraticas/Aula9/codigofonte` - exceto o último, que pode ser acedido através de outro *url* - `https://github.com/uminho-miei-engseg-19-20/EngSeg/blob/master/TPraticas/Aula10/overflow.c`.

LOverflow2 em *C++*

Para o programa em questão verifica-se a existência de *buffer overflow* contudo, compilando o programa em questão sem qualquer opção, não é emitido nenhum *warning*. Ativando as opções *-Wall"* e *-Wextra"*, surge um *warning* que avisa que existe uma variável declarada que não é utilizada, o que de resto é verdade, apesar de não alertar para o problema de *buffer overflow*.

```
LOverflow2.cpp: In function ‘int main()’:
LOverflow2.cpp:9:10: warning: variable ‘tests’ set but not used [-Wun
used-but-set-variable]
     int  tests[10];
          ^~~~~
```

Figura 2.3: *Warning* relativo ao programa *LOverflow2*.

RootExploit em *C*

Relativamente ao programa *RootExploit* existe um *buffer overflow* que está diretamente ligado ao *warning* que é mostrado na *bash* - "a função *gets* é perigosa e não deve ser utilizada". Este *warning* é emitido com e sem opções de *warnings* ativadas e está diretamente ligado a um problema de *buffer overflow*.

```
RootExploit.c: In function ‘main’:
RootExploit.c:10:5: warning: implicit declaration of function ‘gets’;
 did you mean ‘fgets’? [-Wimplicit-function-declaration]
     gets(buff);
     ^~~~
     fgets
/tmp/ccz2G5sV.o: In function `main':
RootExploit.c:(.text+0x37): warning: the `gets' function is dangerous
 and should not be used.
```

Figura 2.4: *Warning* relativo ao programa *RootExploit*.

0-simple em *C*

Este programa tem o mesmo comportamento e problema que o programa anterior.

```
0-simple.c: In function ‘main’:
0-simple.c:16:3: warning: implicit declaration of function ‘gets’; di
d you mean ‘fgets’? [-Wimplicit-function-declaration]
   gets(buffer);
   ^~~~
   fgets
/tmp/ccRkTyrP.o: In function `main':
0-simple.c:(.text+0x3e): warning: the `gets' function is dangerous an
d should not be used.
```

Figura 2.5: *Warning* relativo ao programa *0-simple*.

2-functions em *C*

Este programa é um exemplo claro de que os *warnings*, por vezes também surgem desnecessariamente, ou seja, o compilador não compreende a intenção do utilizador e faz advertências a

algo que o utilizador propositadamente forçou, como é se apresenta o caso a seguir.

```
2-functions.c:26:53: warning: format ‘%p’ expects argument of type ‘v
oid *’, but argument 2 has type ‘int (*)()’ [-Wformat=]
       printf("calling function pointer, jumping to %p\n", fp);
                                                     ~^
```

Figura 2.6: *Warning* relativo ao programa *2-functions*.

overflow em *C*

Para terminar esta demonstração de exemplos práticos da utilidade dos *warnings*, utilizou-se o programa *overflow*. Este é um dos exemplos mais flagrantes da necessidade de programar seguindo boas práticas, isto porque o *warning* em questão apenas aparece se a opção *-Wextra"* for ativa, caso contrário a compilação é feita sem qualquer problema.

```
overflow.c: In function ‘vulneravel’:
overflow.c:7:23: warning: comparison between signed and unsigned inte
ger expressions [-Wsign-compare]
         for (i = 0; i < x; i++) {
                       ^
overflow.c:8:31: warning: comparison between signed and unsigned inte
ger expressions [-Wsign-compare]
                 for (j = 0; j < y; j++) {
                                 ^
overflow.c: In function ‘main’:
overflow.c:16:2: warning: ‘matriz’ is used uninitialized in this func
tion [-Wuninitialized]
  vulneravel(matriz, 5000000000, 5000000000, 0);
  ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Figura 2.7: *Warnings* relativos ao programa *overflow*.

Como podemos ver na figura 2.7, o último *warning* simplesmente indica o facto da variável em questão não ser inicializada na função onde ocorre, no entanto, os dois anteriores estão diretamente relacionados com a ocorrência de um *segmentation fault* que acontece se os valores dados às variáveis x e y forem superiores aos limites das variáveis i e j. O *warning* alerta precisamente para a comparação perigosa de uma variável *signed integer* com outra *unsigned integer*.

2.1.8 *Frama-C*

A verificação formal de programas é algo que está a ganhar cada vez mais relevo e importância nas empresas de *software*, obrigando-as a testar e executar os seus programas em ferramentas que assegurem o bom funcionamento dos mesmos e que mostram que cumprem aquilo para que foram feitos. Por esse motivo, surge aqui esta ferramenta - o ***Frama-C*** [11]- que analisa programas escritos em linguagem *C* de acordo com os objetivos que o seu utilizador se propõe a atingir. Esta ferramenta permite, entre outras funcionalidades, declarar invariantes de ciclo (mostrar, recorrendo à logica, aquilo que o ciclo está a fazer), verificar o cumprimento dos limites das variáveis e estabelecer os *inputs* que devem ser dados e os *outputs* que devem ser obtidos, caso existam.

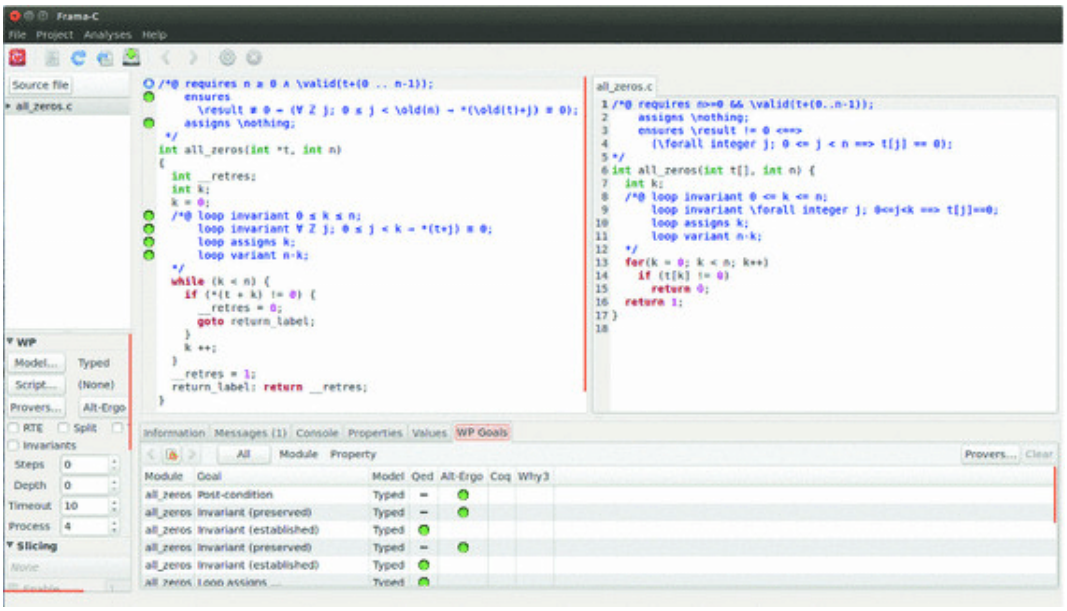


Figura 2.8: Frama-C.

2.2 Java

2.2.1 O que é o Java

O Java é uma linguagem de programação baseada em C/C++ e foi criada por James Gosling. Formalmente anunciada em 1995, o Java é uma linguagem de programação orientada a objetos que é amplamente usada para o desenvolvimento de sites e aplicações. Foi desenvolvida visando a portabilidade e as aplicações Java podem ser encontradas em vários sistemas, como, computadores, telemóveis, etc.

Diferente das linguagens de programação modernas, que são compiladas para código nativo, esta linguagem é compilada para um *bytecode* que é interpretado por uma máquina virtual (Java Virtual Machine - JVM). O Java é rápido, seguro e confiável.

Máquina Virtual Java

- Garbage collector
- Class Loaders
- Faz a gestão a Memória
- Verificador de bytecodes
- Gerenciador de Segurança

Características da Linguagem

- Fortemente tipada
- Não permite aritmética de apontadores
- Exige que variáveis locais sejam atribuídas antes da compilação

- Implementa a verificação de limites de vetores

2.2.2 Funcionamento do Interpretador e Compilador

Os programas Java seguem a linha tradicional de tradução e execução, tal como a linguagem C também o faz. No entanto, o Java foi criado com objetivos distintos das outras linguagens, nomeadamente: funcionar o mais rápido possível e em qualquer plataforma, mesmo que isto implique que tenhamos um declínio no tempo de execução.

A primeira diferença que podemos notar para o processo de tradução e execução das linguagens de programação é que no Java temos que ao invés de compilar para a linguagem de montagem assembly de um computador destino, tem-se uma compilação para instruções fáceis de interpretar: bytecode Java. Em Java, a compilação é instantânea traduzindo bytecodes para código máquina, sendo este executa posteriormente. Isto melhora o tempo de execução do programa.

A Java Virtual Machine ou JVM é um interpretador Java que carrega e executa as aplicações Java que estão em bytecodes, convertendo esses bytecodes em código executável de máquina. Um interpretador nada mais é do que um programa que simula um conjunto de instruções. Dessa forma, não temos uma etapa de montagem separada. A JVM também possui outras funções como o gerenciamento de aplicações na medida em que eles são executados.

A portabilidade é a grande vantagem dessa interpretação. Os mais diversos tipos de dispositivos hoje possuem uma versão da JVM para executar os programas Java, desde os telemóveis até as páginas web. Por este motivo é que o Java é considerado independente de plataforma, pois basta possuirmos uma versão da JVM para os programas escritos em Java poderem funcionar em qualquer plataforma de hardware e software.

Compiladores Just-In-Time foram o próximo passo no desenvolvimento do Java. Compiladores Just-In-time são compiladores que traduziam enquanto o programa estava a ser executado. Esses compiladores nada mais fazem do que traçar o perfil do programa em execução para descobrir onde estão os métodos principais do programa, e depois compilam os mesmos para o conjunto de instruções nativo em que a máquina virtual está a executar. A parte compilada é salva, de modo que a execução possa ser feita mais rapidamente da próxima vez em que ele for executado.

Este equilíbrio entre interpretação e compilação evoluiu bastante com o passar do tempo, de modo que os programas Java executados com frequência sofrem pouquíssimo trabalho extra da interpretação.

2.2.3 Problemas Mais Comuns nas Outras Linguagens

- **Input Validation Error** - Erro devido à passagem de um parâmetro num formato não esperado e não tratado.
- **BoundaryConditionError** - Erro devido a uma tentativa de leitura/escrita num endereço inválido ou esgotamento de um recurso do sistema.
- **BufferOverflow**
- **Access Validation Error** - Erro devido à invocação de operações em objetos fora do domínio de acesso.
- **ExceptionalConditionError** - Erro devido ao não tratamento de uma exceção gerada por um módulo, um dispositivo ou entrada de dados do usuário.

- **CrossSiteScripting** - Inserção de Scripts e codigo HTML em páginas de uma rede local.
- **SQLInjection** - Inserção de comandos SQL através de concatenação de strings usando os separadores de comandos.

2.2.4 Exemplo noutra Linguagem

BufferOverflow

Nesta situação são ultrapassados os limites de um array para executar um código malicioso.
Teoria envolvida?
Ultrapassa-se os limites de um array, alocado dinamicamente numa subrotina, que esteja na pilha de execução até atingir a posição de memória onde está armazenado o endereço de retorno dessa subrotina, que pode ser modificado para apontar para um código malicioso a ser executado.
Porque não afeta aplicações Java?
Os limites dos array são sempre verificados.

FormatString

Pretende-se:

- Conseguir informações sobre os conteúdos dos endereços de memória que normalmente não são acessíveis
- Conseguir uma "shell" de um super-utilizador a partir de uma exceção provocada pela entrada de uma string mal intensionada

Teoria envolvida?
Passar uma string formatada intensionamente para ler e escrever na memória.
Exemplo: funções xprintf e seus especificadores de formato (%s, %x, %d, %u).
Porque não afeta aplicações Java?

- O tratamento de exceções é bastante robusto
- Não é possível ler fora dos limites das variáveis

2.2.5 Vulnerabilidades em Java

Exposição dos Bytecodes

- Possui grande quantidade de informações a respeito no código fonte no seu "pool" de constantes
- Fácil Decompilação

Soluções

* Adição de código não funcional ao Código Fonte
* Ofuscação do Código Fonte
* Criptografia dos arquivos compilados (.class)
* Esconder os arquivos compilados (.class)

Objetos String e Garbage Collection

- São imutáveis
- Garbage Collection não controlável pelo utilizador
- Dados sensíveis podem "flutuar"na memória

Soluções

∗ Usar “array” de “char”

∗ Evitar ao máximo o uso desses objetos em dados sensíveis

Literais de String

- São armazenados exatamente iguais no “pool” de constantes dos arquivos compilados (.class)
- Uma simples análise dos “bytecodes” pode fornecer dados sensíveis

Soluções

∗ Evitar o uso de literais no Código Fonte

∗ Usar “array” de “char” sempre que possível

Conclusões

- As questões de segurança em Java devem ser vistas de forma diferente das outras linguagens
- O programador é o maior responsável pelas vulnerabilidades da aplicação

2.3 Compiler Warnings em Java

Code style

Non-static access to a static member - Quando ativado, o compilador emitirá um erro ou um aviso sempre que um campo ou método estático for acedido com um receptor de expressão. Uma referência a um membro estático deve ser qualificada com um nome de tipo.

Method with a constructor name - Nomear um método com um nome de construtor geralmente é considerada má programação. Quando isto acontece, o compilador sinalizará como um erro ou um aviso.

Potential programming problems

Using a char array in string concatenation - Quando ativado, o compilador emitirá um erro ou um aviso sempre que uma expressão char [] for usada nas concatenações de strings.

"olá"+ novo char [] ’w’, ’o’, ’r’, ’l’, ’d’

Hidden catch blocks - localmente numa instrução try, alguns blocos de captura podem ocultar outros, por exemplo:

```
try {
    throw new java.io.CharConversionException();
} catch (java.io.CharConversionException e) {
} catch (java.io.IOException e) {}.
```

Quando ativado, o compilador emitirá um erro ou um aviso para os blocos de captura ocultos correspondentes às exceções verificadas.

Null pointer access - Quando ativado, o compilador emitirá um erro ou um aviso ao encontrar que uma variável local que certamente é nula é desreferenciada.

Comparing identical - Quando ativado, o compilador emitirá um erro ou um aviso se uma comparação envolver operadores idênticos (por exemplo, 'x == x').

Name shadowing and conflicts

Type parameter hides another type - Quando ativado, o compilador emitirá um erro ou um aviso se, por exemplo, um parâmetro type numa classe interna ocultar um tipo externo.

Interface method conflicts with protected 'Object' method - Quando ativado, o compilador emitirá um erro ou um aviso sempre que uma interface definir um método incompatível com um método Object não herdado. Até que esse conflito seja resolvido, essa interface não poderá ser implementada, por exemplo:

```
interface I {
    int clone();
}
```

Unnecessary code

Local variable is never read - Quando ativado, o compilador emitirá um erro ou um aviso sempre que uma variável local for declarada, mas nunca usada na função.

Unused local or private members - Quando ativado, o compilador emitirá um erro ou um aviso sempre que um membro local ou privado for declarado, mas nunca usado na mesma Class.

Generic types

Usage of a raw type - Quando ativado, o compilador emitirá um erro ou um aviso sempre que encontrar um uso de um tipo "bruto"(ou seja, List em vez de List <>).

Annotations

Annotation is used as super interface - Quando ativado, o compilador emitirá um erro ou um aviso sempre que encontrar um tipo implementando uma anotação. Embora seja possível, considera-se uma má prática.

Unhandled token in '@SuppressWarnings' - Quando ativado, o compilador emitirá um erro ou um aviso sempre que encontrar um token não tratado em uma anotação '@SuppressWarnings'.

Unused '@SuppressWarnings' token - Quando ativado, o compilador emitirá um erro ou um aviso sempre que encontrar um token não utilizado na anotação '@SuppressWarnings'.

2.3.1 Warnings na Prática

Um compiler warning é diferente de um compiler error, pois todo o código ainda é compilado com um erro do compilador. Um compiler warning é o Java a informar que devemos examinar melhor alguma coisa.

É preciso saber identificar quando ocorre um compiler warning. Ao compilar uma classe exemplo, os avisos do compilador serão, por exemplo, assim:

```
$ javac *.java
Note: Some input files use unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

Neste exemplo, o Java está basicamente a dizer que sabe que foi usado código antigo e pergunta se queremos saber mais. Se for passada aquela flag, obter-se-á algo como o seguinte. (As mensagens exatas dependerão do compilador utilizado. Por exemplo, em alguns compiladores, receber-se-á um 4º aviso que diz o objeto Unicorn é declarado.)

```
$ javac -Xlint:unchecked *.java
LegacyDragons.java:9: warning: [unchecked] unchecked call to add(E) as a member
of the raw type List
    unicorns.add(new Unicorn());
                ^
  where E is a type-variable:
    E extends Object declared in interface List
LegacyDragons.java:11: warning: [unchecked] unchecked method invocation: method
printDragons in class LegacyDragons is applied to given types
    printDragons(unicorns);
                ^
  required: List<Dragon>
  found: List
LegacyDragons.java:11: warning: [unchecked] unchecked conversion
    printDragons(unicorns);
                 ^
  required: List<Dragon>
  found:    List
3 warnings
```

2.4 Python

2.4.1 O que é o Python

Python é uma linguagem de programação orientada a objetos, de alto nível. A sua simplicidade reduz a manutenção do programa. É capaz de suportar módulos e pacotes e incentiva a

programação modularizada e a reutilização de códigos.

2.4.2 Funcionamento do Interpretador e Compilador

Python é uma linguagem de programação interpretada como também uma linguagem compilada. O **compilador** traduz a linguagem Python para uma linguagem de baixo nível, neste caso, *bytecode*. O **interpretador** executa este bytecode numa Máquina Virtual. A mesma pode ser de vários tipos consoante a implementação. A mais comum é a *CPython* que é a implementação *default* e é escrita em C. Existem ainda outras alternativas como é o caso do **Jython** que é a implementação Python escrita em Java que utiliza a JVM(Java Virtual Machine) e do **IronPython** que é outra implementação Python, escrita em C e que se foca na stack .NET.[1]

Caso existam erros, o compilador gera um relatório de erros e o interpretador interrompe a tradução quando encontra o primeiro erro.

Por norma, o tempo de execução de um código compilado é menor do que um interpretado já que o compilado é inteiramente traduzido antes de sua execução, enquanto que o interpretado é traduzido instrução por instrução.

2.4.3 Vulnerabilidades

O *CNCS* (Centro Nacional de Cybersegurança) alerta para possíveis vulnerabilidades que possam existir. Para o sistema Python, foram identificados pacotes de bibliotecas que contém software malicioso no repositório oficial do Python. Nesses pacotes apenas há ligeiras alterações de nome e,portanto, fazem-se passar por bibliotecas seguras. O uso destas bibliotecas provoca a extração de informação do utilizador e da respetiva máquina, enviando-a para um servidor remoto.[2]

Para além deste, também o sistema *CVE* (Common Vulnerability Exposure) é responsável por identificar e corrigir vulnerabilidades encontradas. No caso do Python, explicamos de seguida algumas vulnerabilidades a ele associadas :[3]

- Ao manipular indevidamente alguns arquivos *pickle*, um atacante pode usar esse bug para consumir memória através de negação de serviço (DoS). Este bug afeta apenas o Ubuntu 16.04 e o Ubuntu 18.04.
- Um atacante pode enganar o Python enviando *cookies* para o domínio errado, porque o Python,ao manipular as cookies, validou incorretamente o domínio.
- O Python manipulou incorretamente a criptografia Unicode durante a normalização do NFKC. Um atacante pode usar isso para obter informações confidenciais(atacando assim a confidencialidade).
- O Python manipulou incorretamente o esquema *local_file*, o que poderia ser usado por um atacante remoto para contornar os mecanismos da *blacklist*.
- entre outros.

2.4.4 Warnings

Relativamente aos *warnings*, são geralmente emitidos quando o encerramento do programa não é garantido. Em python, os *warnings* são emitidos recorrendo-se à função **warn()**. As mensagens de warning são gravadas no *sys.stderr* podendo a sua disposição ser alterada, isto é,

ignorar todos os avisos ou transformá-los em exceções.Warnings específicos que são repetidos são, por norma, omitidos.

Existem dois estados no controlo de warnings. O primeiro, é feita uma determinação se uma mensagem deve ou não ser emitida que é controlada por um filtro. Este filtro é uma sequência de regras e ações correspondentes que podem ser adicionadas ao mesmo chamando a função **filterwarnings()** e podem ser redefinidas para o seu estado original recorrendo à função **resetwarnings()**. O segundo estado é que caso uma mensagem seja emitida, é formatada e impressa usando um "gancho"configurável pelo utilizador. A impressão da mensagem é feita recorrendo à função **showwarning ()** e para a sua formatação é chamada a função **formatwarning ()**.[4]

Existem várias categorias de warnings que são mencionadas e explicadas de seguida:

- **Warning:** Esta é a classe base de todas as classes das categorias dos warnings. É uma subclasse de exceção.
- **UserWarning:** É a categoria default para *warn()*.
- **DeprecationWarning:** Categoria base para warnings sobre recursos não aprovados quando os warnings são destinados a outros programadores de Python (ignorados por default, exceto se forem acionados pelo código em __main__).
- **SyntaxWarning:** Categoria base para warnings sobre recursos sintáticos duvidosos.
- **RuntimeWarning:**Categoria base sobre recursos de tempo de execução duvidosos.
- **FutureWarning:**Categoria base para warnings sobre recursos não aprovados quando os warnings são destinados a utilizadores finais de aplicações escritas em Python.
- **PendingDeprecationWarning:** Categoria base para warnings sobre recursos que não serão aprovados no futuro(ignorados por default).
- **ImportWarning:**Categoria base para warnings que são acionados quando se importa um determinado módulo(ignorados por default).
- **UnicodeWarning:**Categoria base para warnings relacionados com o Unicode.
- **BytesWarning:** Categoria base para warnings relacionados com as classes *bytes* e *bytesarray*.
- **ResourceWarning:**Categoria base para warnings relacionados com o uso de recursos.

Por outro lado, também existem os filtros de warnings que decidem se os warnings são ignorados, exibidos ou transformados em erros. Conceptualmente,este filtro mantém uma lista ordenada de especificações de filtro que estão na forma **(ação, mensagem, categoria, módulo, lineno)**. De seguida, são explicados os termos anteriores:

- **ação**
  - "default":imprime a primeira ocorrência de warnings correspondentes a cada local (módulo + número da linha).
  - "error": transforma warnings em exceções.
  - "ignore": nunca imprime warnings correspondentes.

  - "always": imprime sempre warnings correspondentes.
  - "module":imprime a primeira ocorrência onde cada módulo em que warning é emitido (independentemente do número da linha).
  - "once": imprime apenas a primeira ocorrência dos warnings correspondentes, independentemente da localização.

- **mensagem:** é uma string que contém uma expressão regular que deve corresponder ao ao início do warning. A expressão é compilada para sempre fazer distinção entre maiúsculas e minúsculas.
- **categoria:** é uma classe (uma subclasse de Warning)que deve corresponder a uma subclasse da categoria de warning.
- **módulo:** é uma string que contém uma expressão regular que deve corresponder ao nome do módulo. A expressão é compilada para fazer distinção entre maiúsculas e minúsculas.
- **lineno:** é um inteiro que deve corresponder ao número da linha em que o warning ocorreu ou 0 para corresponder a todos os números de linha.

Estes filtros de warning são inicializados pelas opções -W passadas para como argumento e chama a função **filterwarnings**().

2.4.5 Warnings na Prática

Para demonstrar o que foi dito na secção anterior, podem ser observados alguns exemplos que lançam warnings.

Gerar Warnings

A maneira mais fácil de lançar um warning é chamar **warn()** com a mensagem como argumento.

```
import warnings

print ('Before the warning')
warnings.warn('This is a warning message')
print ('After the warning')
```

É possível definir o filtro para gerar um erro no UserWarning. Conseguimos observar a exceção.

```
thug80:TP2$ python -W "error::UserWarning::0" warnings_warn.py
Before the warning
Traceback (most recent call last):
  File "warnings_warn.py", line 4, in <module>
    warnings.warn('This is a warning message')
UserWarning: This is a warning message
```

Filtrar com Padrões

Para filtrar com base no conteúdo do texto da mensagem são necessárias regras mais complexas, por isso utiliza-se a função **filterwarnings ()**.

```
import warnings

warnings.filterwarnings('ignore', '.*do not.*',)

warnings.warn('Show this message')
warnings.warn('Do not show this message')
```

O padrão contém "do not", mas a mensagem usa "Do not". O padrão corresponde porque a expressão regular é sempre compilada para procurar correspondências sem distinção entre maiúsculas e minúsculas.

```
thug80:TP2$ python warnings_filterwarnings_message.py
warnings_filterwarnings_message.py:5: UserWarning: Show this message
  warnings.warn('Show this message')
```

Warnings Repetidos

Normalmente, a maioria dos warnings aparecem apenas na primeira vez em que ocorrem num determinado local, sendo o mesmo definido como a combinação de número do módulo e linha.

A ação "once"pode ser usada para eliminar instâncias da mesma mensagem de diferentes locais.

```
import warnings

warnings.simplefilter('once', UserWarning)

warnings.warn('This is a warning!')
warnings.warn('This is a warning!')
warnings.warn('This is a warning!')
```

```
thug80:TP2$ python warnings_once.py
warnings_once.py:5: UserWarning: This is a warning!
  warnings.warn('This is a warning!')
```

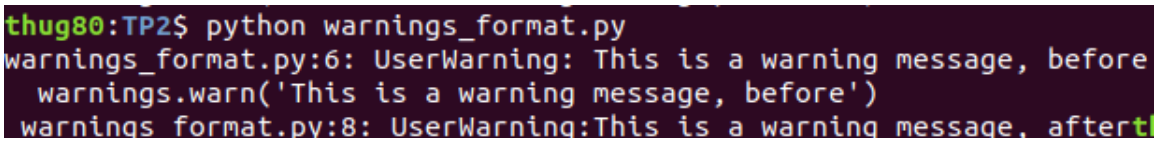
Formatting

Caso não haja problema de os warnings serem direcionados para o stderr, mas o utilizador não gosta da formatação é possível substituir a função **formatwarning()**.

```
import warnings

def warning_on_one_line(message, category, filename, lineno, file=None, line=None):
    return ' %s:%s: %s:%s' % (filename, lineno, category.__name__, message)

warnings.warn('This is a warning message, before')
warnings.formatwarning = warning_on_one_line
warnings.warn('This is a warning message, after')
```


thug80:TP2$ python warnings_format.py
warnings_format.py:6: UserWarning: This is a warning message, before
warnings.warn('This is a warning message, before')
warnings_format.py:8: UserWarning:This is a warning message, after

3. Conclusão

Tudo aquilo que se relatou neste documento não é mais do que um conjunto de boas práticas que desde sempre devem ser implementadas em qualquer tipo de projeto. Contudo, no caso em questão, as mesmas focam a área da engenharia de segurança e exemplificam tipos de medidas que se podem tomar para evitar determinados problemas extremamente simples e que podem implicar custos extremamente avultados para as organizações que neles incorrem - relembrem-se o caso *heartbleed*.

A utilização e implementação de boas práticas na programação deve indiscutivelmente, ser uma preocupação daqueles que querem construir *software* fidedigno e funcional, visto que a inclusão de mais um linha de código que verifica uma determinada variável, pode fazer a diferença entre uma empresa ter ou não que sofrer consequências gravíssimas por falhas de *software*.

É agora mais do que evidente que os argumentos que comummente se ouvem são infundados e já se podem refutar com situações práticas. Há relatos de várias organizações que começaram a investir nesta área após verificarem a presença de erros banais nos seus sistemas, alguns que resultaram em repercussões económicas impensáveis.

A compilação de programas deve, por todos os motivos vistos anteriormente, ter em atenção todos os *erros* e *warnings* que são fundamentais para que os programas construidos sejam fiáveis e não originem erros durante a execução dos mesmos. É uma mais valia e é importante lembrar que não há qualquer custo monetário necessário para obter esses avisos, pelo que essa não pode ser uma desculpa válida. Além dessas, existem outras formas, também elas gratuitas, de verificar programas disponíveis para várias linguagens.

Bibliografia

[1] https://pt.wikipedia.org/wiki/Java_(linguagem_de_pr ograma%C3%A7%C3%A3o)

[2] https://www.devmedia.com.br/processo-de-traducao-e-execucao-de-programas-java/26915

[3] https://www.linuxtopia.org/online_books/eclipse_document ation/eclipse_java_development_guide/topic/org.eclipse.jdt.doc.user/reference/preferences/java/compiler/eclipse_java_ref-preferences-erro rs-warnings.html

[4] https://books.google.pt/books?id=Vwf9CgAAQBAJpg=PA115&lpg=PA115dq=t%C3%A9cnicas+de+Co PTsa=X&ved=2ahUKEwi65JqP0Y7pAhUS9IUKHZQOATcQ6AEwAXoECAgQAQ#v=onepageqf=false

[5] Cncs.gov.pt. 2020. Alerta De Vulnerabilidade - Bibliotecas Python. [online] Available at: <https://www.cncs.gov.pt/recursos/alertas-de-seguranca/alerta-de-vu lnerabilidade-bibliotecas-python/?fbclid=IwAR3Dh8rpT5tRMjqhxE948EXo 39vWaE4bBFX2fOl2oFMldBCsS4LlylQNyfo> [Accessed 3 May 2020].

[6] Cncs.gov.pt.2020. Alerta De Vulnerabilidade - Bibliotecas Python. [online] Available at:<https://www.cncs.gov.pt/recursos/alertas-de-seguranca/alerta-de-vulnerabilidade-bibliotecas-python/?fbclid=IwAR3Dh8rpT5tRMjqhxE948EXo39vWaE4bBFX2fOl2oFMldBCsS4LlylQN yfo> [Accessed 3 May 2020].

[7] Ubunlog. 2020. Python Tenía Varias Vulnerabilidades Que Incluso Podían Drenar La Batería. [online] Available at: <https://ubunlog.com/python-tenia-varias-vulnerabilidades-que-incluso-podian-drenar-la-bateria-de-nuestros-equipos/> [Accessed 4 May 2020].

[8] Docs.python.org. 2020. Warnings — Warning Control — Python 3.8.3Rc1 Documentation. [online] Available at: <https://docs.python.org/3/library/warnings.html> [Accessed 3 May 2020].

[9] GNU - Warnings - https://gcc.gnu.org/onlinedocs/gcc/Warning-Options.html

[10] GNU - GDB - https://www.gnu.org/software/gdb/

[11] Frama-C - https://www.gnu.org/software/gdb/