Artefato WTICG'23 Apêndice: Artigo #235074: Instrumentação de programas binários legados para compatibilização com Intel CET

Éverton C. Araújo[1], **Mateus Tymburibá**[1], **Andrei Rimsa**[1]

[1]Departamento de Computação
Centro Federal de Educação Tecnológica de Minas Gerais (CEFET-MG)
Belo Horizonte – MG – Brasil

araujocostaeverton@gmail.com,{mateustymbu,andrei}@cefetmg.br

Resumo. *Mecanismos de proteção contra ataques que alteram o fluxo de execução de programas têm sido desenvolvidos por fabricantes de semicondutores. A tecnologia CET (Control-flow Enforcement Technology), criada recentemente pela Intel, é um exemplo promissor desse tipo de proteção. Para usá-la, programas devem incorporar uma nova instrução adicionada à arquitetura x86. Portanto, essa solução não se aplica a programas que não podem ser recompilados. Este trabalho propõe a utilização de técnicas de instrumentação para identificar as posições de um binário onde essa nova instrução deve ser adicionada. Posteriormente, ela é embutida nesses pontos identificados, utilizando uma técnica de remendo que garanta a integridade do executável.*

1. Selos Considerados

Solicitamos avaliação para atribuição dos selos Disponíveis e Funcionais.

2. Informações básicas

Neste artefato apresentamos os procedimentos a serem executados para replicação do experimento executado durante o desenvolvimento do trabalho. O programa de teste foi escrito em Linguagem C (*loopJump.c*) e instrumentado para compatibilização com a tecnologia Intel CET. Os arquivos necessários para execução deste experimento se encontram no repositório público do Github (araujoec/instr-to-cet-compat[1]).

2.1. Requisitos

1. Sistema operacional *Linux (ou derivado do Unix)*;
2. Compilador GCC (versão utilizada: 11);
3. *objdump*, *dd* e *GDB* (utilitários nativos do *Linux/Unix*);
4. CFGgrind[2];
5. *dot*[3].

3. Instalação *CFGgrind*

A ferramenta *CFGgrind* possui um tutorial próprio de instalação e teste que pode ser encontrado em seu repositório público do Github (link na nota de rodapé abaixo).

[1]https://github.com/araujoec/instr-to-cet-compat
[2]https://github.com/rimsa/CFGgrind
[3]https://graphviz.org/

4. Compilação do programa de exemplo

O programa *loopJump.c* foi compilado com a versão 11 do GCC por meio do seguinte comando no terminal de linha de comandos:

```
$ gcc -fcf-protection=none -o loopJump loopJump.c
```

5. Experimento

O experimento pode ser inteiramente reproduzido pelo terminal de linha de comandos com o auxílio de ferramentas de visualização de imagens e de arquivos de texto.

Para visualizar o arquivo objeto do programa *loopJump.c* e executar a análise estática, a fim de identificar as instruções de desvio indireto, deve-se utilizar o seguinte comando:

```
$ objdump -d loopJump
```

A saída desse comando pode ser comparada ao conteúdo do arquivo "*loopJump-dump.txt*", disponível no repositório. No endereço *114c* da saída deste comando deve-se encontrar a instrução de desvio indireto com a instrução *jump* ("*jmp *%rax*"). Nesse caso, o *objdump* já consegue calcular e identificar o endereço de destino armazenado no registrador *rax*, conforme pode ser observado na linha anterior ao endereço mencionado. Entretanto, ainda assim, a análise dinâmica pode ser realizada.

Para gerar o arquivo de mapeamento das instruções *assembly* com o nome "*loopJump.map*", utiliza-se o seguinte comando:

```
$ cfggrind_asmmap ./loopJump > loopJump.map
```

Com esse arquivo é possível gerar outros dois arquivos, de extensões *.cfg* e *.dot*, que são utilizados para construir a imagem do grafo de fluxo de controle da função *main* por meio do seguinte comando:

```
$ valgrind -q --tool=cfggrind --cfg-outfile=loopJump.cfg \
    --instrs-map=loopJump.map --cfg-dump=main ./loopJump
```

A partir do arquivo de extensão *.dot* gerado, é possível então construir a imagem com o comando:

```
$ dot -Tpng -o loopJump.png cfg-0x109129.dot
```

A imagem construída permite a visualização do endereço de destino da instrução de salto indireto *jump* e, a partir dela, criar os arquivos de remendo e trampolim. Esses arquivos devem ser construídos após a conversão das instruções *assembly* para seus valores binários correspondentes, representados em hexadecimal. Para isso, os comandos a seguir criam os arquivos binários de remendo e de trampolim, respectivamente:

```
$ echo -e -n     "\xf3\x0f\x1e\xfa\xe9\x28\x00\x00\x00\x90" \
    > remendo.bin

$ echo -e -n "\x83\x7d\xfc\x00\x7e\xe3\x83\x6d\xfc\x01" \
    "\xe9\xca\xff\xff\xff"  > trampolim.bin
```

O utilitário *dd* permite a sobrescrita do binário *loopJump* nos endereços desejados e pode ser feito com os seguintes comandos:

```
$ dd if=remendo.bin of=loopJump bs=1 count=10 seek=4404 conv=notrunc

$ dd if=trampolim.bin of=loopJump bs=1 count=15 seek=4453 conv=notrunc
```

Por fim, para verificar a execução do programa passando por cada instrução após a instrumentação, o utilitário *GDB* pode ser utilizado com o seguinte comando:

```
$ gdb loopJump
```

O comando a seguir exibe a função *main* em instruções *assembly*:

```
$ disassemble /r main
```

Um *breakpoint* pode ser colocado no endereço *[address]* desejado por meio do comando:

```
$ break *0x[address]
```

Para visualizar cada instrução, a seguinte sequência de comandos deve ser executada:

```
$ set disassemble-next-line on
$ show disassemble-next-line
$ run
```

E, finalmente, para iterar pelas instruções, utilize o seguinte comando:

```
$ stepi
```