ADICIONANDO E ESTENDENDO INSTRUÇÕES DO DISTORM3

24/12/2013

Conforme discutido em alguns dos artigos anteriores, o software Open Source oferece como vantagem a facilidade de modificação de seu código para adaptação às necessidades encontradas. Isso também se apresenta como uma vantagem ao pesquisador, pois pode demonstrar seu conhecimento e poupar muito tempo de trabalho ao encontrar soluções quase prontas para os problemas que enfrentará. Com o intuito de demonstrar a teoria aqui exposta, vou modificar um disassembler muito popular e explicar o que foi feito. Isso permitirá a outros pesquisadores fazerem adaptações similares, melhorando o software em si, além de melhorar o tempo de aprendizado do mesmo.

A ideia veio a tona quando eu vi um artigo escrito pela Microsoft [1] sobre um novo malware que utilizava algumas instruções não documentadas (para ser sincero, eu li o artigo quando eles o lançaram, mas esqueci completamente do mesmo até que vi uma alteração no código do Pev [2] , outro software Open Source, para detectar a técnica utilizada pelo malware).

O que chamou minha atenção imediatamente quando eu li o artigo pela primeira vez foi o fato de que, apesar dos disassemblers demonstrados, e os debuggers também não conseguirem traduzir corretamente a instrução (por não ser suportada). Ela não tinha um efeito real na visão do resto do código (não causava nenhum desalinhamento, não fazia o debugger se perder, nem o disassembler, portanto não poderia de fato ser uma técnica de ofuscação ou de anti-disassembler/debugger).

Dado que sou um grande fan do diStorm (e do Open Source, como já mencionei em outros artigos), eu decidi testar o código e ver os resultados (não abordarei aqui neste post como compilar o diStorm, como o mesmo funciona ou extensões possíveis, mas fiquem a vontade para me enviar dúvidas):

```
diStorm version: 3.3.0
bits: 32
filename: ../examples/linux/fpu.out
origin: 00000000
00000000 (01) d9 DB 0xd9
00000001 (02) d8df FCOMP ST7
00000003 (01) df DB 0xdf
00000004 (01) df DB 0xdf
00000005 (01) df DB 0xdf
```

00000006 (01) df DB 0xdf
00000007 (01) df DB 0xdf
00000008 (01) df DB 0xdf
00000009 (01) df DB 0xdf
0000000a (01) df DB 0xdf
0000000b (01) df DB 0xdf
0000000c (02) 8bec MOV EBP, ESP

Como podemos ver o diStorm também não reconhece as instruções (ele as substitui por
DB por pensar se tratar de dados), então, teremos de estender o mesmo para que funcione corretamente. Após ler o documento de desenvolvimento do mesmo [3], estava claro para mim as decisões de design e como elas complicavam as mudanças no software (apesar de fazerem sentido considerando as vantagens em termos de performance). Mas não existia referência alguma quanto a como adicionar novas instruções ou ampliar instruções correntes.

Na verdade, minha tarefa foi muito mais simples do que eu pensei inicialmente, dado que existia software que acompanhava o projeto (não documentado) que o desenvolvedor do mesmo (Gil) proveu:

disOps (desenvolvido em python)

Se você editar o arquivo x86sets.py (disOps/x86sets.py) verá uma sequência de chamadas a uma função (Set()). Essa sequência está na ordem dos bytes do opcode, então, para nosso caso, precisamos adicionar uma instrução (FSTPNCE) e estender outra (FSTP) - Uma boa dica é sempre olhar em instruções similares antes de tentar criar sua própria modificação:

Set("df //07", ["FISTP"], [OPT.FPUM64], IFlag.MODRM_REQUIRED)
Set("df //e0", ["FNSTSW"], [OPT.ACC16], IFlag.INST_FLAGS_NONE)

No arquivo original (mostrado acima parcialmente), você verá que o diStorm não suporta o opcode DFDF para a FSTP, portanto, eu mudei para:

Set("df //07", ["FISTP"], [OPT.FPUM64], IFlag.MODRM_REQUIRED)
Set("df //df", ["FSTP"], [OPT.FPU_SSI], IFlag._32BITS)
Set("df //e0", ["FNSTSW"], [OPT.ACC16], IFlag.INST_FLAGS_NONE)

Está bem simples o que isso faz:

Define o opcode DF, seguido de outro DF como a instrução FSTP. Ela possui dois registradores FPU como parte da mesma (ST0, ST7). As flags definem o que o resto da

instrução recebe de parâmetro (no nosso caso, o segundo registrador que é o ST7).

Também precisamos adicionar a nova instrução (FSTPNCE):

```
Set("d9 //c9", ["FXCH"], [], IFlag.INST_FLAGS_NONE)
Set("d9 //d0", ["FNOP"], [], IFlag.INST_FLAGS_NONE)
Set("d9 //e0", ["FCHS"], [], IFlag.INST_FLAGS_NONE)
```

Acima temos novamente a sequência do arquivo original (e a instrução desejada não existe), então mudamos para:

```
Set("d9 //c9", ["FXCH"], [], IFlag.INST_FLAGS_NONE)
Set("d9 //d0", ["FNOP"], [], IFlag.INST_FLAGS_NONE)
Set("d9 //d8", ["FSTPNCE"], [OPT.FPU_SI, OPT.FPU_SI],
IFlag.INST_FLAGS_NONE)
Set("d9 //e0", ["FCHS"], [], IFlag.INST_FLA

GS_NONE)
```

Desta vez temos o opcode D9 seguido de D8 e a mesma recebe dois registradores FPU como parâmetro (ST0, ST0).

Devemos também mudar em disOps/disOps.py um comentário que existe no final do arquivo (próximo do fim da função main):
DumpMnemonics()

Mudamos para (removemos o comentário):
DumpMnemonics()

Quando rodamos o disOps, teremos como output um arquivo chamado output.txt. Este arquivo é usado para substituir o src/insts.c (após os includes, que na versão que utilizei vai da linha 29 até o final):

```
vi src/insts.c
:29,$ d -> deleta da linha 29 até o fim do arquivo
:x -> sai e salva
cat disOps/output.txt >> src/insts.c ->
```

Para anexar ao src/insts.c o conteúdo (poderia ter feito de dentro do vi)

O disOps também gera um arquivo disOps/defs.txt que contém uma enum que devemos inserir em include/mnemonics.h (é um typedef de _InstructionType).

Se compilarmos, veremos que o disassembly ainda está incorreto, dado que a instrução FSTPNCE aparecerá como FCHS. Isso acontece pois o arquivo insts.c apenas possui os ponteiros e flags para as estruturas de dados, mas não possui o mnemonico em si. Isso é parte do mnemonics.c:

const unsigned char _MNEMONICS[] =

Aqui temos para cada índice da tabela o tamanho da string que representa o mnemônico, seguido do mesmo, por exemplo:

"\x09" "UNDEFINED\0"

Dado que o índice é ordenado pelo opcode (o tipo de estrutura de dados chama-se Tries), precisamos posicionar o FSTPNCE antes do FCHS:

"\x04" "FNOP\0" "\x07" "FSTPNCE\0" "\x04" "FCHS\0" "\x04" "FABS\0" "\x04" "FTST\0" "\x04" "FXAM\0"

Aqui nós colocamos:

"\x07" -> a string FSTPNCE possui 7 bytes

Compilando e testando finalmente temos o resultado desejado:

```
diStorm version: 3.3.0
bits: 32
filename: ../examples/linux/fpu.out
origin: 00000000
00000000 (02) d9d8 FSTPNCE ST0, ST0
00000002 (02) dfdf FSTP ST0, ST7
00000004 (02) dfdf FSTP ST0, ST7
00000006 (02) dfdf FSTP ST0, ST7
00000008 (02) dfdf FSTP ST0, ST7
0000000a (02) dfdf FSTP ST0, ST7
0000000c (02) 8bec MOV EBP, ESP
```

REFERÊNCIAS:

[1] Radu, Daniel. "Investigation of a new undocumented instruction trick". Site:http://blogs.technet.com/b/mmpc/archive/2013/06/24/investigation-of-a-ne.... Last Accessed: 11/05/2013.

[2] Merces, Fernando. "PEV - PE File Analysis Toolkit". Site: https://github.com/merces/pev. Last Accessed: 11/05/2013.

[3] Dabah, Gil. "diStorm internals". Site: https://code.google.com/p/distorm/wiki/diStorm_Internals. Last Accessed: 11/05/2013.

APÊNDICE A (the patch):

Dado que criar o patch em si envolve apenas alguns passos, mas a geração do output e a substituição do arquivo insts.c gera uma grande mudança (>15000 linhas), estou incluindo aqui apenas
o link para download do patch (para quem não quer seguir o artigo, mas quer as mudanças). O patch será incorporado ao SVN do diStorm3 de qualquer forma, então, se não for pelo estudo, recomendo que não se preocupe com isso:
http://www.kernelhacking.com/rodrigo/files/distorm3_fstpnce.patch

Sobre o autor

Rodrigo Rubira Branco (BSDaemon), atua como pesquisador sênior no centro de excelência em segurança da Intel . Foi fundador do projeto Dissect || PE de análise de malware e palestrante em diversas conferências nacionais e internacionais, tais como Blackhat, Defcon, Hack in The Box, XCon e Hackito.

Membro do comitê técnico de diversas conferências (Blackhat Brasil, Hackito e Nosuchcon, por exemplo) também foi palestrante principal (keynote) em eventos fora do Brasil e em território nacional. É organizador do evento Hackers to Hackers (H2HC), maior e mais antigo evento de pesquisas em segurança da informação na América Latina. Atuou em diversas empresas, tais como Check Point (como Chief Security Research) e Qualys (como Diretor de Pesquisas de Vulnerabilidades e Malware). Também é conselheiro do Instituto Coaliza.

Em 2011 foi homenageado pela Adobe como um dos contribuidores principais em vulnerabilidades nos produtos da empresa. Brasileiro convicto (apesar de ter morado em Israel, Dubai e atualmente nos Estados Unidos), é membro do Comitê Técnico da RENASIC, ligada ao Centro de Defesa Cibernética (CDCiber) do Departamento de Defesa Brasileiro.