RevEngE: Extensões de Engenharia Reversa para o GDB

Marcus Botacin[1], **Lucas Galante**[2], **Paulo Lício de Geus**[2], **André Grégio**[1]

[1] Federal University of Paraná (UFPR)

{mfbotacin, gregio}@inf.ufpr.br

[2]University of Campinas (Unicamp)

{galante, paulo}@lasca.ic.unicamp.br

Abstract. *Reverse engineering binaries is an essential task in the security field, both for legitimate software validation as well as for malware analysis. Whereas GDB is a powerful tool for legitimate program analysis, it presents some drawbacks for analyzing armored malware code. To overcome these drawbacks, we propose RevEngE, a Reverse Engineering Engine that adds malware analysis capabilities to GDB.*

Resumo. *A engenharia reversa de binários é uma tarefa essencial no campo da segurança, tanto para a validação de aplicações legítimas quanto para a análise de códigos maliciosos. Ainda que o GDB seja uma solução poderosa para a análise de aplicações benignas, este apresenta limitações para lidar com aplicações maliciosas especialmente desenvolvidas para impedir a engenharia reversa. Para superar estas limitações, propomos RevEngE, um motor de engenharia reversa que adiciona capacidades de análise de malware ao GDB.*

1. GDB: Recursos & Limitações

O *GNU Debugger* (GDB) [GNU 2019] é a solução de depuração mais utilizada no ambiente GNU/Linux, sendo, por isto, escolhida por nós como base de desenvolvimento de uma solução de engenharia reversa de *malware*.

O GDB possui todas as funções essenciais para a inspeção de aplicativos (legítimos ou maliciosos), tais como pontos de parada (*breakpoints*), execução passo a passo (*step-by-step*) e a reconstrução de contexto através da inspeção de valores armazenados em registradores ou em memória, sendo, portanto, muito utilizado na engenharia reversa de aplicações em geral [tdumitra 2015].

Apesar de seus inúmeros recursos, o GDB não foi projetado para lidar especificamente com códigos maliciosos, apresentando diversas limitações quando da inspeção destes. Destacamos, em particular, 3 limitações principais: (i) dificuldades para lidar com binários sem símbolos (*stripped*); (ii) ausência de suporte nativo para o *tracking* de múltiplos fluxos de execução; (iii) possibilidade de evasão de análise pela detecção do ambiente de depuração.

Binários *Stripped*. Códigos maliciosos são geralmente compilados sem símbolos de qualquer natureza, o que dificulta a reconstrução do contexto original. Em particular, muitos binários maliciosos não possuem um símbolo para ponto de entrada (*entry point*)

da função principal (*main*) claramente definido. A identificação deste símbolo é um pré-requisito para a criação de *breakpoints* automáticos no GDB. Nossa solução se propõe a identificar automaticamente os endereços dos pontos de entrada e fornece-los ao GDB para que um analista possa inspecionar um código malicioso da mesma forma que inspeciona aplicações tradicionais.

Múltiplos Fluxos de Execução. Códigos maliciosos frequentemente criam múltiplos caminhos de execução para dificultar a análise, criando um problema conhecido como explosão de caminhos [Chen et al. 2014]. A execução de aplicações no GDB segue apenas um dos caminhos possíveis, o que limita o trabalho do analista. Nossa solução se propõe a permitir ao analista seguir diferentes fluxos de execução além do fluxo principal seguido pelo GDB, ampliando, assim, as capacidades de análise deste.

Identificação do GDB. Códigos maliciosos frequentemente se recusam a executar em ambientes de análise para impedir que o analista entenda o funcionamento e desenvolva contra-medidas para o exemplar de *malware*. Uma forma comum de se implmentar este tipo de técnica de anti-análise é detectar a presença de um depurador. Como o GDB se baseia no suporte do sistema operacional para sua operação, através da biblioteca *ptrace*, sua detecção é trivial. Nossa solução se propõe a prover mecanismos para impedir a detecção do GDB por exemplares de *malware* sob análise.

2. RevEngE: Arquitetura & Implementação

Nesta seção, apresentamos a arquitetura proposta e a implementação do RevEngE.

2.1. Arquitetura

A principal vantagem de basear nossa solução no GDB é minimizar a curva de aprendizado imposta aos analistas, que já estão familiarizados com o GDB. Desta forma, nossa solução foi projetada de modo a atuar sob o GDB, estendendo este através de diversos comandos a serem acessados via *prompt* da solução, tal qual comandos nativos.

Este tipo de estensão é possibilitada pela existência de um interpretado *Python* dentro do GDB [Python.org 2017], que permite que comandos e resultados de inspeções sejam interceptados e tratados pela linguagem de alto nível antes de serem exibidos ao usuário. Este tipo de abordagem é frequentemente utilizado para expandir o GDB para fins de segurança ofensiva [gef , Pwndbg], sendo a nossa proposta a primeira a propor tal extensão para fins de engenharia reversa.

2.2. Implementação

Uma vez que o desenvolvimento de métodos *Python*, de uma maneira geral, tal qual o tratamento de *strings* e o desenvolvimento de *parsers*, é muito provavelmente familiar aos analistas interessados neste trabalho, nos limitamos a descrever em detalhes apenas a implementação dos métodos responsáveis por superar os 3 desafios de análise anteriormente descritos.

Identificação do *Entry Point*. No caso de binários compilados com base na *libc*, o endereço do ponto de entrada é apontado pelo primeiro parâmetro do ponto de entrada da própria *libc*, como mostrado no Código 1.

```
1 __libc_start_main (main=<value optimized out>, argc=<value optimized
      out>, ubp_av=<value optimized out>,
2      init=<value optimized out>, fini=<value optimized out>, rtld_fini
          =<value optimized out>,
3      stack_end=0x7fffffffdc38) at libc-start.c:258
```

Código 1. LibC. O primeiro argumento aponta para o ponto de entrada da aplicação.

Portanto, para se identificar o ponto de entrada da aplicação, pode-se utilizar o próprio GDB para se interromper a execução do código quando da chamada da *libc* e então inspecionar o valor de memória apontado pelo primeiro argumento desta. RevEngE automatiza esta tarefa da seguinte maneira:

1. Identificando automaticamente o endereço da *libc* a partir do cabeçalho do binário ELF.
2. Definindo um ponto de parada no endereço identificado.
3. Executando o código até o ponto de parada especificado.
4. Obtendo o endereço apontado pelo primeiro parâmetro.
5. Definindo um ponto de parada no endereço apontado.
6. Executando o código até o novo ponto de parada.
7. Devolvendo o controle da inspeção ao analista quando este ponto de parada é atingido.

Alteração do fluxo de execução. Toda árvore de decisão se origina em uma instrução de desvio (*branch*) estas, por sua vez, dependem das *flags* setadas no registrador de *flags* do processador. Desta forma, uma instrução `jz` (*jump when zero*), por exemplo, toma caminhos diferentes se a *flag* `ZF` (*Zero Flag* estiver ou não definida. Portanto, uma forma de forçar o GDB a seguir diferentes caminhos é alterar as flags definidas no momento da inspeção, como exemplificado pelo Código 2.

```
1 output = gdb.execute("set␣$eflags|=0x%x" % self.flag_map[flag],
      to_string=True)
```

Código 2. Inverter direção do salto. O registrador de flags é alterado de acordo com um mapa de possiveis valores de flags para cada instrução.

***Bypass* de verificações de inspeção**. A forma mais simples e comum de se identificar a execução dentro do GDB é tentar anexar o GDB ao próprio processo e verificar se esta falha com o *status* de GDB já anexado, como mostrado no Código 3.

```
1 if (ptrace(PTRACE\_TRACEME, 0, NULL, 0) == -1)
```

Código 3. Identificação de Ptrace. Rotinas são ignoradas para permitir a continuidade da inspeção.

Para evitar que exemplares de *malware* evadam a análise através desta técnica, RevEngE se beneficia do mecanismo anteriormente apresentado e automaticamente inverte o *branch* de forma que o *malware* não seja capaz de identificar sua operação dentro do GDB, permitindo, assim, que o analista prossiga com os procedimentos de análise.

3. Experimentação

Nesta seção, descrevemos como RevEngE pode ser obtido e utilizado.

Download. RevEngE é uma solução de código aberto e pode ser baixado em: `https://github.com/marcusbotacin/Reverse.Engineering.Engine`

Instalação. RevEngE pode ser instalado através do comando `source install.sh`

Execução. Os principais comandos para a operação do RevEngE são:

- `revtest`. Executa automaticamente todas as operações e rotinas de teste.
- `revstart`. Inicia manualmente o RevEngE.
- `revstop`. Finaliza manualmente o RevEngE.
- `revstep`. Avança manualmente para a próxima instrução.
- `revinv`. Inverte manualmente a direção da instrução de salto.
- `reventry`. Identifica manualmente o endereço de entrada.

Demonstração. RevEngE será demonstrado através da engenharia reversa de exemplares de *malware* reais, como exemplificado no vídeo: **link**.

Agradecimentos

Este trabalho é financiado pelo Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq, Bolsa de Doutorado, processo 164745/2017-3) e pela Coordenação para o Aperfeiçoamento Pessoal do Ensino Superior and (CAPES, Projeto FORTE, Programa de Ciências Forenses, 24/2014, processo 23038.007604/2014-69).

Referências

Chen, B., Zeng, Q., and Wang, W. (2014). Crashmaker: An improved binary concolic testing tool for vulnerability detection. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, SAC '14, pages 1257–1263, New York, NY, USA. ACM.

gef. Gef - gdb enhanced features for exploit devs & reversers. `https://github.com/hugsy/gef`.

GNU (2019). Gdb: The gnu project debugger. `https://www.gnu.org/s/gdb/`.

Pwndbg. Pwndbg. `https://github.com/pwndbg/pwndbg`.

Python.org (2017). gdb support. `https://devguide.python.org/gdb/`.

tdumitra (2015). Using gdb for reverse engineering. `http://users.umiacs.umd.edu/~tdumitra/courses/ENEE757/Fall15/misc/gdb_tutorial.html`.