Segurança
Patrulha do código

O Linux oferece algumas ferramentas sofisticadas para entender como o malware pode escapar por entre as linhas de código de um aplicativo desavisado.

por Andrew Henderson

O perigo potencial do malware é uma preocupação para muitos usuários de computadores. Os desktops, smartphones e até mesmo os dispositivos de rede inteligentes que temos em casa ou no escritório são potencialmente vulneráveis a milhares de *rootkits*, *spywares* e *trojans*. Na medida em que desenvolvedores de software antimalware criam novos métodos para descobri-los e bloqueá-los, os autores de malwares criam novos métodos para contornar essas salvaguardas.

Usuários Linux têm apreciado por muito tempo a atenção mínima que recebem dos autores de malware. O malware não existe para a plataforma Linux, pois encontra dificuldades em infectar a maioria dos sistemas baseados em UX. Uma das principais contribuições do software de código aberto é a velocidade com que os exploits causadores de bugs são notados, diagnosticados e corrigidos. Esta resposta rápida limita o intervalo de tempo em que o software malicioso pode explorar essas vulnerabilidades. A menos que o malware possa de alguma forma enganar um usuário com uma conta de superusuário para instalá-lo, ou o sistema não esteja devidamente configurado ou atualizado, ele muitas vezes terá dificuldade de infectar sistemas Linux.

Os usuários de Windows não tiveram tanta sorte. Existem no mundo mais de um bilhão de PCs executando várias versões do Windows, e tal base instalada de software torna-o um alvo muito tentador para os autores de malware. O grande tamanho e complexidade da base de código do Windows, bem como os atrasos na disponibilização de patches para correções, oferece ampla oportunidade para as vulnerabilidades serem descobertas e exploradas pelos autores de malware.

Surpreendentemente, o Windows encontra um par de aliados incomuns na luta contra o malware: o Linux e o software de código aberto. Uma série de ferramentas Linux tem sido utilizadas para realizar análises sofisticadas de malware do Windows e para melhorar a qualidade do software. Os desenvolvedores usam essas ferramentas para criar um código melhor que é mais seguro, e os pesquisadores as utilizam para entender melhor como o malware explora o código inseguro.

Estas ferramentas de análise de software caem em duas categorias gerais: estática e dinâmica. Ferramentas de análise estática examinam o código fonte ou binário de um programa para entender melhor como o programa está estruturado e como ele reage a entradas específicas de dados. As ferramentas de análise dinâmica assistem o software à medida em que este é executado para observar diretamente seu comportamento em tempo de execução. Cada um destes métodos de análise tem seus pontos fortes e fracos, mas ambos são muito úteis na investigação de software desconhecido. Neste artigo, apresentaremos alguns dos conceitos utilizados nestas análises e mostraremos uma variedade de ferramentas de código aberto que pesquisadores de segurança usam atualmente para analisar e compreender a infecção por malware.

Análise estática

A análise estática de software existe há muito tempo, e o leitor provavelmente realizou muitas delas por si mesmo. Se compilou o código fonte, por exemplo, certamente realizou uma análise estática; cada compilador analisa enquanto compila para descobrir os nomes de símbolos duplicados, atribuições incompatíveis de entrada, variáveis não inicializadas ou linhas inacessíveis de código.

O frontend Clang [1] para a infraestrutura do compilador LLVM contempla um poderoso analisador estático, o que lhe permite fornecer mensagens de erro muito detalhadas e avisos do compilador. A passagem de análise padrão do GCC (que é um analisador bastante poderoso) pode ser aumentada através do uso de plugins de análise [2] para o compilador. O suporte a tais plugins está disponível nas versões 4.5 e posteriores do GCC. Cada programa contém grupos de instruções que são executadas sequencialmente, se não ocorrerem exceções ou trocas de contexto. A execução do programa flui através dessas instruções como um único bloco lógico. Na terminologia do compilador, este grupo lógico de instruções é conhecido como um bloco básico (BB). Quando a execução de um BB termina, o controle flui para o início de outro BB e a execução continua. O fim de um BB é geralmente onde um salto ou desvio condicional ocorre.

A análise estática tenta descobrir todos esses BBs e determinar onde é possível controlar o fluxo de um BB para outro. Essas informações são coletadas e representadas como um gráfico de controle de fluxo (CFG) que descreve o amplo cenário de como um programa é executado. Para melhor ilustrar isso, a **listagem 1** exibe uma função C simples, e a **figura 1** mostra a CFG que descreve os BBs e o fluxo de controle da função. A análise estática desta função quebra o código em três BBs: A, B e C. O controle deve sempre fluir de A para C, porque esses BBs são os pontos de entrada e saída para a função.

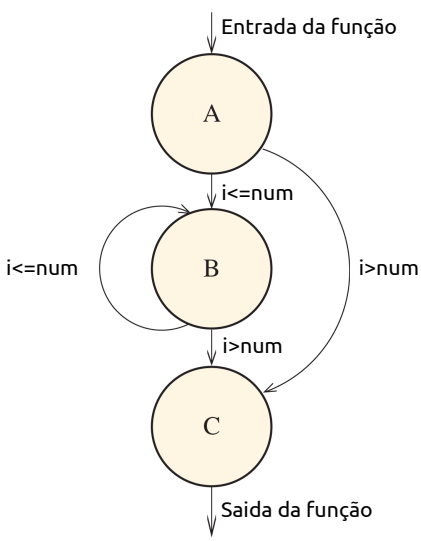

Figura 1 O gráfico de controle de fluxo (CFG) gerado para a função C simples presente na **listagem 1**.

B pode nunca ser executado, ou pode executar várias vezes. Esses comportamentos podem ser observados ao examinarmos o CFG sem nunca realmente executarmos a função. Embora o controle de fluxo desta função de exemplo seja muito simples de visualizar, funções extremamente complexas são muito mais difíceis de analisar por questões de segurança, sem gerar um gráfico de controle de fluxo correspondente.

Todos os dados que entram em um sistema devem vir de alguma fonte, como um arquivo, comunicações de rede ou um periférico como um teclado ou mouse. Os dados que se originam a partir de uma fonte que é considerada não confiável (por exemplo, uma conexão de rede) também são considerados não confiáveis. Alguns BBs também contêm instruções que implementam um evento sensível, como uma chamada de sistema (alocação de memória, leitura ou gravação de arquivos, comunicação por sockets etc.). Se os dados não confiáveis influenciarem estes eventos, existe a possibilidade de explorar o software e comprometer o sistema. Os CFGs simplificam a tarefa de descobrir quais BBs geram dados não confiáveis, quais contêm eventos sensíveis e quais controlam os fluxos que conectam os dois. Uma finalidade principal da análise estática de malware é descobrir como os dados não confiáveis podem potencialmente propagar código sensível.

Listagem 1: Uma função C simples

```
01 int simple(int num)
02 {
03 /* Start block A */
04 int i;
05 printf("Block A\n");
06 /* End block A */
07 for (i = 1; i <= num; i++)
08 {
09 /* Start block B */
10 printf("Block B\n");
11 /* End block B */
12 }
13 /* Start block C */
14 printf("Block C\n");
15 return(0);
16 /* End block C */
17 } Andrew Henderson.
```

Quando o código fonte está disponível, realizar uma análise estática é simples. Descrições completas e inequívocas de tipos de dados estão disponíveis, e é trivial identificar as chamadas de função e estruturas de controle de fluxo. Mas, o código fonte para o malware muitas vezes não está disponível, o que significa que os binários devem ser analisados estatisticamente para determinar seu comportamento. Embora isto possa ser feito através da análise do binário instrução por instrução (o que muitos estudos de malware já fizeram), é muitas vezes mais útil analisar uma representação de código fonte em maior nível do binário ao fazer sua decompilação.

Gerar uma representação de linguagem de alto nível de um recurso binário não é trivial. Os decompiladores devem determinar quais segmentos do binário contêm código e dados, e devem reconstruir tipos de dados complexos. Nenhum decompilador gera código fonte que seja tão claro quanto o código original no qual o binário foi compilado, e a assistência manual

de um especialista é muitas vezes necessária para completar e esclarecer a tradução. Existem vários decompiladores de código aberto para Linux, mas o decompilador Boomerang [3] é provavelmente o melhor disponível atualmente. O Boomerang está em desenvolvimento ativo e faz um trabalho relativamente bom de gerar uma representação de código C para um determinado binário.

Arquivos de classe Java baseados em bytecode são muito mais simples de decompilar porque contêm uma grande quantidade de informações contextuais (por exemplo, herança e digitação de dados) em suas instruções de bytecode. Esta informação contextual torna a decompilação bytecode mais simples e menos ambígua do que a decompilação de binários nativos. Muitos pesquisadores de segurança usam o pacote de ferramentas em código aberto Soot [4] para decompilar arquivos de classe Java, embora uma variedade de outras ferramentas estejam disponíveis no Linux para executar a decompilação Java. A ferramenta Dare (*Dalvik Retargeting*) [5] pode ser usada para converter arquivos em formato `dex` da máquina virtual Dalvik do Android para arquivos de classe Java, que podem então ser decompilados por ferramentas feitas para Java.

Análise dinâmica

A análise estática é muito poderosa, mas não pode fazer um trabalho muito bom de captura de problemas relacionados à simultaneidade em software multithreaded. Se assim fosse, já existiriam compiladores para detectar e avisar os desenvolvedores sobre as condições da execução e outros problemas relacionados à simultaneidade presentes no código. Também é comum para o malware usar binários compactados ou criptografados, o que acrescenta uma camada de ofuscação que frustra a decompilação e a análise estática.

Devido a essas limitações, por vezes a única maneira de analisar corretamente o malware é observar seu comportamento em tempo de execução. Esta observação é chamada de análise dinâmica. A análise dinâmica é realizada em muitos níveis de detalhe diferentes, que vão desde assistir a um evento ocasional de um processo individual à execução do kernel instrução por instrução até passar por cada processo em todo o sistema. A análise dinâmica é extremamente poderosa, mas também tem algumas grandes desvantagens. A análise só é realizada nas porções de um programa que a executa, de modo que o comportamento malicioso do malware deve ser executado enquanto a análise é executada. Caso contrário, a análise resultante é inútil. Há também um impacto negativo significativo sobre o desempenho do sistema, pois a execução do monitoramento da instrumentação cada vez que uma instrução de CPU é executada pode ocasionar a lentidão de um programa ou a execução de sistema por uma ordem de magnitude ou mais.

Em 2005, um grupo de pesquisadores propôs um novo método de análise dinâmica chamado "*dynamic taint analysis*" ("análise dinâmica de danos"), que aborda o problema do rastreamento de dados não confiáveis. Em vez de tentar executar uma análise estática em cada malware "*zero-day*" assim que ele surge (o que é uma tarefa bastante árdua, com milhares de novos malwares surgindo todos os dias) e, em seguida, criar um software para se proteger contra isso, por que não tentar proteger os sistemas automaticamente rastreando dados não confiáveis de forma dinâmica, uma vez que se propagam por todo o sistema? Se a execução for interrompida antes que os dados não confiáveis sejam usados para executar uma ação perigosa, o malware é neutralizado sem saber os detalhes exatos de implementação e design.

A análise dinâmica de danos considera uma fonte de dados não confiável como uma fonte corrompida. Todos os dados criados por uma fonte corrompida estão "contaminados" pela marcação dos dados com um identificador "*taint tag*". Essas taint tags indicam de qual fonte corrompida os dados foram originados. Locais onde os dados corrompidos podem fazer algo potencialmente perigoso são chamados de "*taint sinks*". Os dados corrompidos são monitorados na medida em que são criados, copiados, transformados, movidos e destruídos ao copiar e destruir as taint tags associadas aos dados. Se os dados corrompidos chegam a um taint sink, a execução do sistema é interrompida e a análise forense pode começar.

É perigoso armazenar uma taint tag dentro do mesmo espaço de memória como dados comuns porque outras escritas de dados podem substituir a taint tag e corrompê-la ou destruí-la. Também é difícil armazenar informações da taint tag juntamente com dados corrompidos, se os dados forem movidos para um registro da CPU, porque não há nenhum espaço extra no registo para suportar a taint tag. Para evitar esses problemas, todas as taint tags são armazenadas em "shadow memories". Para cada local potencial onde os dados corrompidos possam residir (memória RAM,

registradores da CPU, buffers de dispositivos, periféricos etc.), existem *shadow memories* análogas e separadas que possuem as taint tags associadas.

Uma única taint tag representa o defeito associado a um único pedaço de dados. A granularidade do pedaço único de dados depende das necessidades de análise. Em muitos casos, representar um único byte de dados com cada taint tag geralmente é bom o suficiente. Para uma análise muito detalhada, pode ser necessário criar uma taint tag para cada bit de dados. O número de diferentes tipos de taint tags também podem variar. Para algumas análises, ter um único tipo de taint tag para cada pedaço de dado corrompido para indicar que "este não é um dado confiável" é suficiente. Em outros casos, ter um tipo diferente de taint tag para cada fonte corrompida é necessário. O caso mais extremo seria uma taint tag única para cada bit de dados não confiável que entra no sistema, embora esta configuração exija uma grande quantidade de shadow memory para implementar.

A **figura 2** mostra um exemplo de como o defeito propaga o uso de taint tags. Neste exemplo, cada parte dos dados é um byte, e a taint tag associada a esses dados é também um único byte. Uma operação está para ser executada, o que adicionará os dados localizados no endereço 0x1000 aos dados localizados no endereço 0x1002. O resultado será armazenado no endereço 0x1000. Os dados no endereço 0x1002 estão corrompidos porque existe uma taint tag na posição da shadow memory associada ao endereço 0x1002. Uma vez que a operação de inclusão se completa e o resultado é armazenado em 0x1000, a taint tag na shadow memory se propaga a partir do endereço 0x1002 para 0x1000 porque os dados corrompidos no 0x1002 influenciaram os dados no 0x1000. Os dados contidos em 0x1000 estão agora corrompidos, e qualquer operação usando estes dados resultará em uma propagação ainda maior da taint tag.

A propagação dos dados contaminados da fonte corrompida para uma taint sink revela uma grande quantidade de informação sobre como o malware compromete os sistemas. Não só a fonte de dados maliciosos pode ser determinada, como quaisquer cópias dos dados também são conhecidas por buscar as taint tags apropriadas. Na verdade, esta forma de análise de danos tornou-se tão útil para a segurança que as linguagens Perl, PHP, Lua, Python e Ruby possuem suporte embutido para ela. Desenvolvedores conscientes das questões que envolvem segurança, e que usam essas linguagens, têm utilizado a análise de danos (tainting) para checar a segurança de seus softwares nos últimos anos.

Um bom exemplo de uma ferramenta de análise dinâmica de nível de processo é o Pin, da Intel [6]. O Pin é um framework para a instrumentação binária de processos em plataformas x86. Ele é anexado a um processo em execução, da mesma forma como faz o depurador GDB e, em seguida, avança na execução do programa de uma instrução de assembly por vez. Plugins customizados para o Pin, chamados de "Pintools", são desenvolvidos em C/C++ com o uso de bibliotecas e scripts de construção distribuídos com o Pin. O Pintools implementa chamadas de retorno de instrumentação que disparam antes da execução de cada instrução enquanto o binário executa. Esta abordagem permite a criação de ferramentas de análise que são usadas para tarefas como log de instrução, profiling, análise de danos e para verificar se os argumentos de chamadas de sistema são válidos.

Criar ferramentas que realizam a análise dinâmica em nível de sistema, acrescentando instrumentação lógica para VM hypervisors (Xen, VMware) ou emuladores do sistema inteiro (QEMU, Bochs) oferece várias vantagens. É perigoso executar análise de software dentro do mesmo ambiente onde o malware executa pois o malware pode detectar o software de análise e, em seguida, atacá-lo ou se esconder dele. Ao utilizar os recursos de virtualização de VMs para isolar a lógica de análise do malware, a análise torna-se inviolável. Usar uma VM também oferece o benefício de sermos capazes de observar o estado dos registradores da CPU, memórias stack e heap, e periféricos, a qualquer momento durante a análise.

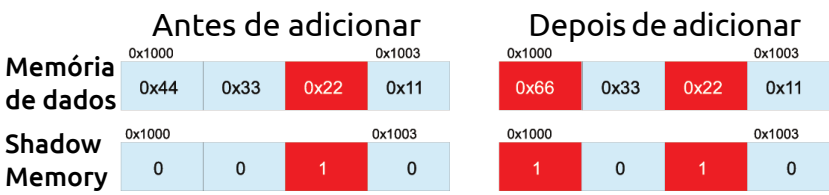

Figura 2 Os estados de memória de dados e sua shadow memory correspondente antes e depois de uma operação de inclusão.

Preparação com DECAF

Uma plataforma de análise dinâmica completa em recursos e sistema é o *Dynamic Executable*

Code Analysis Framework (DECAF) [7]. O DECAF é um software de código aberto que está em desenvolvimento ativo por pesquisadores e aficionados. Usando o emulador QEMU para todo o sistema [8] em seu núcleo, o DECAF ostenta um rico conjunto de recursos de análise, incluindo um plugin de API para a criação de ferramentas de análise personalizada, de sobrecarga de execução mais baixa do que a maioria das outras plataformas de análise completa, e a capacidade de executar e analisar os ambientes mais recentes de sistemas operacionais como Windows 7 e 8.

A **figura 3** mostra a arquitetura geral do DECAF. O DECAF executa o sistema operacional convidado que está sendo analisado dentro da VM convidada. Uma CPU emulada e uma unidade de gerenciamento de memória (MMU) são usadas pela VM convidada. Os dispositivos de hardware que o sistema operacional convidado usa, (por exemplo, disco rígido, NIC, teclado e mouse) são emulados em software pelo DECAF e apresentados à VM convidada como dispositivos de hardware. Do ponto de vista do sistema operacional convidado, ele está sendo executado em um hardware real e desconhece que esteja sendo monitorado.

A arquitetura do sistema DECAF também mostra algumas das shadow memories usadas para armazenar taint tags. Todas essas shadow memories estão localizadas fora da VM convidada para evitar a adulteração da taint tag pelo sistema operacional convidado. A MMU da VM convidada usa um RAM shadow para manter as taint tags para quaisquer dados corrompidos localizados na memória RAM física da VM. Alguns dispositivos periféricos, como a NIC, possuem um shadow buffer encobre buffers de transmissão/recepção interna periférica.

O armazenamento secundário possui um shadow buffer próprio para marcar quais dados no disco estão corrompidos. Mesmo que a leitura e escrita de dados corrompidos de/para arquivos não seja de interesse em uma determinada análise completa, manter este shadow buffer é muito importante por conta da paginação de memória do sistema operacional. Se uma página da RAM física é retirada do disco pelo kernel, as taint tags associadas a esses dados devem ser copiadas para o shadow buffer de armazenamento secundário. Quando trocam os dados do disco para a memória RAM física, as taint tags associadas devem ser copiadas para a shadow RAM. Caso contrário, informações corrompidas serão perdidas quando a RAM física for retirada do disco.

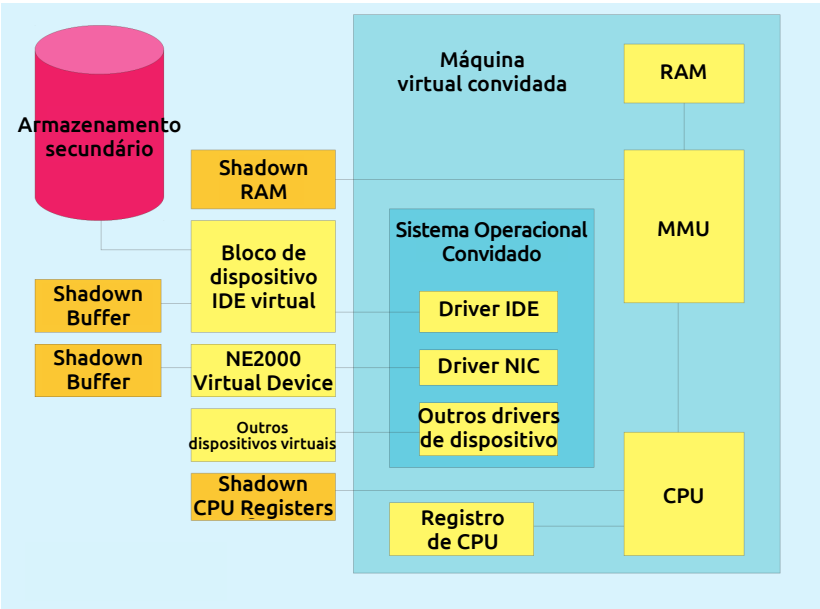

Figura 3 A arquitetura geral do sistema DECAF.

O futuro

À medida em que as estações de trabalho e servidores tornam-se cada vez mais rápidos, a complexidade das análises que podem ser realizadas no software só irá aumentar. A disponibilidade de muitas ferramentas de análise de software em código aberto tem trazido uma enorme vantagem tanto para pesquisadores como para desenvolvedores na luta contra o malware e na criação de software mais seguro. A pesquisa de malware é um campo em crescimento e os esforços para desenvolver melhores ferramentas de análise irão beneficiar os desenvolvedores de software que utilizam diversas linguagens no Linux. E talvez, apenas talvez, poderá beneficiar alguns usuários do Windows também. ■

Mais informações

[1] Clang: http://clang.llvm.org
[2] Plugins GCC: http://gcc.gnu.org/onlinedocs/gccint/Plugins.html
[3] Boomerang: http://boomerang.sourceforge.net
[4] Soot: http://www.sable.mcgill.ca/soot
[5] Dare: http://siis.cse.psu.edu/dare
[6] Pin: http://software.intel.com/en-us/articles/pintool
[7] DECAF: https://code.google.com/p/decaf-platform
[8] QEMU: http://wiki.qemu.org/Main_Page

Consulte o principal
Partner Magento
do Brasil.

e-smart

MAGENTO GOLD SOLUTION PARTNER

Capitais e Regiões Metropolitanas
4003-1137

Demais Regiões
0800-606-4210

www.e-smart.com.br