Stack Overflow

Bruna Almeida Osti
Rafael Cortez Sanches
17 de Junho de 2020

Resumo

Esse ensaio tem por objetivo apresentar o conceito de estouro de *buffer* (*buffer overflow*) e detalhes técnicos das vulnerabilidades que o originam. Formas de explorá-las são apresentadas, bem como uma coletânea de incidentes relacionados. Por fim, são elencadas medidas úteis para prevenir a exploração dessas vulnerabilidades.

1 Introdução

O estouro de buffer acontece quando um programa excede o uso de memória especificado para a aplicação pelo sistema operacional, passando a escrever no setor de memória contíguo. Esse tipo de problema é causado por softwares mal construídos que não preveem controles de memória, podendo ter diferentes deficiências como: estouros de pilha, corrupção de heap, bugs nos formatos das strings, entre outros (1).

Um programa bem construído deveria estipular um tamanho máximo para dados recebidos e garantir que esses valores não sejam superados, pois eles serão armazenados na memória de acesso aleatório, em um setor conhecido como buffer. Por outro lado, as instruções e dados de um programa em execução são armazenados temporariamente em forma contígua da memória em um setor chamado pilha (stack). Sendo assim, os dados localizados depois do buffer contêm um endereço de retorno (ponteiro de instrução) que permite que o programa continue sendo executado. Se a quantidade de dados é maior que o tamanho do buffer, o endereço de retorno é sobrescrito e o programa passa a ler um endereço de memória inválido, o que gera uma violação no segmento da aplicação (1)(2).

É possível incluir um código no buffer que possibilite a abertura de um interpretador de comandos (como um shell) e permita a alguém mal intencionado controlar esse sistema. Um cibercriminoso com conhecimentos sólidos em gerenciamento de memória e sistemas operacionais consegue facilmente executar esse ataque (2).

Neste trabalho explicaremos conceitos a respeito do estouro de buffer, abordaremos alguns incidentes envolvendo esse tipo de vulnerabilidade e como podemos nos prevenir.

2 Buffer Stack Overflow

Nesta seção abordaremos conceitos necessários para o entendimento do funcionamento de pilhas, como acontece o estouro de pilha e como isso pode ser sinônimo de vulnerabilidades.

2.1 Funcionamento da pilha de memória

A pilha é um mecanismo que os computadores utilizam para passar argumentos e referenciar variáveis de funções locais. É uma maneira de facilitar o acesso de dados locais em uma função específica e passar as informações do chamador da função. Além disso, tem comportamento de buffer, mantendo todas as informações necessárias à função de forma estática apesar dos dados contidos na pilha mudarem (1).

O funcionamento da pilha é controlado por registradores, os principais são:

- **EIP - ponteiro de instruções estendido:** Quando você chama uma função, o ponteiro é salvo na pilha para uso posterior. Quando a função retornar, esse endereço salvo é usado para determinar a localização da próxima instrução executada.
- **ESP - ponteiro de pilha estendido:** Isso aponta para a posição atual na pilha e permite que coisas sejam adicionadas e removidas do empilhe usando operações push e pop ou manipulações diretas de ponteiro de pilha.
- **EBP - O ponteiro base estendido:** Esse registro geralmente permanece o mesmo durante a execução de uma função. Serve como um ponto estático para referenciando informações baseadas em pilha, como variáveis e

dados em um função usando deslocamentos. Esse ponteiro geralmente aponta para o topo da pilha de uma função.

No processador Intel x86 a pilha é a região selecionada pelo *SS registrador de segmento*.O ponteiro de pilha ESP funciona como um deslocamento do segmento base e sempre contém o endereço no elemento superior da pilha. Esse tipo de processador é considerado invertido, pois as pilhas crescem para baixo como descrito na Figura 1 (1).

Quando um item é empurrado para a pilha (PUSH), o ESP diminui e o novo elemento é gravado no lugar resultante. Por outro lado, quando um elemento é retirado da pilha (POP), o elemento é lido a partir do local em que os pontos ESP são aumentados, se movendo em direção ao limite superior e diminuindo a pilha, como mostrado na Figura 1. Vale lembrar que quando o elemento é adicionado ao topo da pilha ele é gravado cada vez mais abaixo de todas as entradas anteriores, por este motivo você poderia sobrescrever o que deve estar nos endereços mais altos incluindo os valores do registrador EIP (1).

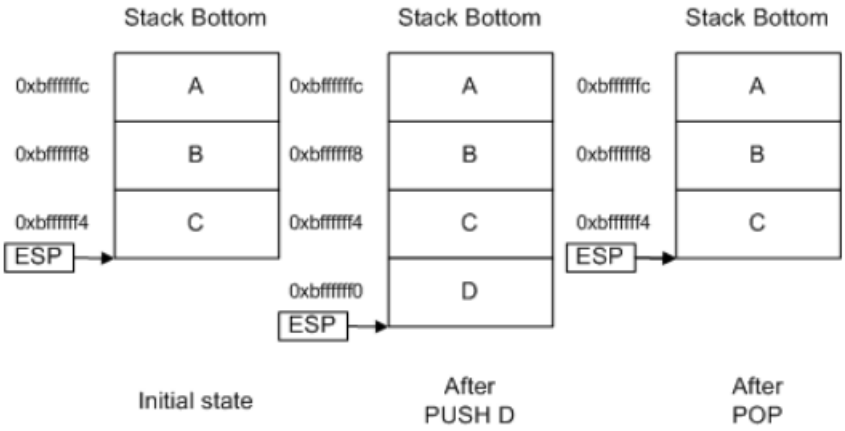

Figura 1: Funcionamento da pilha (1)

2.2 Estouro de Pilha (Stack)

Um estouro de buffer ocorre quando muitos dados são colocados no buffer, isso significa que existe um buffer de tamanho fixo em algum lugar da pilha. Como a pilha cresce e há informações muito importantes armazenadas, o que acontece se você colocar mais dados no buffer alocado da pilha do que ele pode suportar? Como o copo de água, ele transborda e derrama dados adicionais em áreas adjacentes da pilha como descrito na Figura 2 (1).

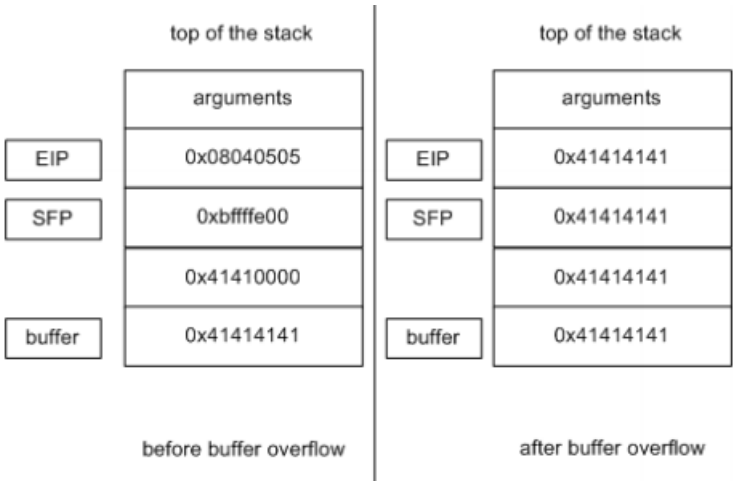

Figura 2: Sobrescrevendo os dados armazenados(1)

3 Exploração da Vulnerabilidade

Como citado anteriormente é possível incluir um código no buffer que possibilite a abertura de um interpretador de comandos (como um shell) e permita a alguém mal intencionado controlar esse sistema, isto acaba sendo uma vulnerabilidade causada pelo estouro da pilha como descrito na Subseção 2.2. Nesta seção veremos algumas técnicas de exploração que são utilizadas.

3.1 Injeção de buffer

O primeiro passo para criar uma exploração é encontrar uma maneira de obter seu buffer grande no buffer transbordável. Normalmente, esse é um processo simples, automatizando o preenchimento de um buffer pela rede ou gravando um arquivo que é lido posteriormente por o processo vulnerável. Às vezes, no entanto, levar seu buffer para onde ele precisa estar pode ser um desafio em si (1).

3.1.1 Otimização do vetor injetado

O vetor de injeção é o código necessário para controlar o ponteiro de instruções na máquina remota, no qual depende da máquina e do destino (1).

O ponto principal do vetor de injeção é executar o payload. O payload por outro lado, é muito parecida com um vírus: deve funcionar em qualquer lugar, a qualquer momento, independentemente de como foi injetada na máquina remota (1).

3.1.2 Determinando a localização do payload

É necessário se preocupar com o tamanho do payload e como o vetor de injeção interage com ela, pois se a carga começar antes do vetor de injeção, é necessário garantir que não colidam. Se o fizerem, é necessário incluir um salto na carga útil para pular o código de injeção (1).

Alguns lugares geralmente utilizados:

- Arquivos no disco, que são carregados na memória
- Variáveis de ambiente controladas por um usuário local
- Variáveis de ambiente passadas em uma solicitação da Web (comum)
- Campos controlados pelo usuário em um protocolo de rede

Depois de injetar a carga, a tarefa é simplesmente fazer com que o ponteiro da instrução carregue o endereço da carga.

3.2 Métodos para execução do payload

3.2.1 Direct Jump (Adivinhando Offsets)

O salto direto significa que você disse ao seu código de estouro para ir diretamente para um local específico na memória. Ele não usa truques para determinar a verdadeira localização da pilha na memória (1).

Os problemas dessa abordagem são que primeiramente, o endereço da pilha pode conter um caractere nulo; portanto, toda a carga útil precisará ser colocada antes do injetor. No entanto em máquinas UNIX, o endereço da pilha geralmente não contém um caractere nulo, tornando este o método de escolha para estouros UNIX (1).

3.2.2 Call Register

Essa técnica é popular nas explorações do Windows, porque existem muitos comandos em endereços fixos no Kernel32.dll. Esses pares podem ser usados em quase qualquer processo normal. Como eles fazem parte da DLL da interface do kernel, eles normalmente estarão em endereços fixos, que você pode codificar. No entanto, eles provavelmente diferirá entre as versões do Windows e possivelmente dependerá de qual Service Pack é aplicado (1).

Se um registro já estiver carregado com um endereço que aponte para o payload, o atacante simplesmente precisa carregar o EIP em uma instrução que executa uma "CALL EDX"ou "CALL EDI"ou equivalente dependendo do registrador desejado (1).

4 Incidentes Envolvendo Buffer Overflow

Essa seção reúne uma coletânea de incidentes notórios envolvendo a exploração de *buffer overflow*, ordenados de forma cronológica.

4.1 O Internet Worm de Robert Morris (1988)

Além de explorar falhas no modo de debug da aplicação *sendmail* para sistemas UNIX, o Internet Worm de Morris também fez uso de uma vulnerabilidade de buffer overflow no programa *fingerd*, um *daemon* para processar requisições geradas pelo programa *finger*.

Em seu código, o *fingerd* não limitava o tamanho da entrada fornecida pelo usuário, mas mesmo assim utilizava um buffer capaz de armazenar até 512 bytes de dados. Fornecendo uma entrada maior que esse limite, Morris conseguiu provocar um estouro de pilha e injetar um comando para ser executado pela máquina alvo. (3)

Sabendo da configuração da pilha de execução em sistemas BSD, Morris conseguiu substituir o endereço de retorno na pilha do *fingerd* para forçar a máquina a executar uma sessão do shell para ele. Uma vez que os administradores rodavam o *fingerd* como root, isso dava privilégios para execução de código como superusuário ao atacante. (4)

4.2 Code Red Worm (2001)

Uma vulnerabilidade de *buffer overflow* foi identificada pela equipe da Microsoft em uma das extensões de seu servidor Web comercial, o Internet Information Services (IIS). Descrita em um boletim de segurança da companhia, essa vulnerabilidade consistia em suprir um URL de entrada com tamanho suficientemente grande para estourar o buffer de entrada, permitindo tomar controle total da máquina, uma vez que servidores Web geralmente são executados com privilégios. (5)

O boletim de segurança foi publicado em junho, sugerindo que todos os administradores aplicassem o *patch* de segurança fornecido pela Microsoft o mais rápido possível. No entanto, quando Marc Maiffret e Ryan Permeh lançaram um *worm* para explorar essa vulnerabilidade em julho, muitos servidores ainda não haviam sido atualizados. O *malware* conhecido como *Code Red* infectou mais de 359.000 hospedeiros em menos de 14 horas. (6)

5 Prevenindo Ataques de Buffer Overflow

Uma das soluções mais indicadas para desenvolvedores é a de evitar o uso de bibliotecas e funções inseguras. No entanto, conforme os sistemas comerciais aumentam de tamanho, a identificação de vulnerabilidades no código se transforma em uma tarefa cada vez mais difícil. Uma saída é o hábito de se praticar revisão de código nas organizações, fazendo com que essas inspeções sejam habituais e aplicadas sempre que um novo fragmento de código for incorporado ao sistema. Ferramentas de análise de código fonte também podem ajudar a encontrar potenciais vulnerabilidades.

Funcionalidades de compiladores também podem ajudar a prevenir estouros de pilha, como é o caso do *Stackguard* para o compilador GNU gcc. Ele usa um *arbitrary canary*, representado por uma palavra da arquitetura alvo, o qual é inserido entre as variáveis da pilha e o endereço de retorno. Dessa forma, quando um estouro de pilha tentar sobrescrever o endereço de retorno, ele pode invalidar o *arbitrary canary*, permitindo que o programa identifique em tempo de execução que a pilha da função executada foi comprometida. No entanto, esse método não é a prova de falhas, uma vez que atacantes podem tentar contornar o *arbitrary canary*. (3)

6 Considerações Finais

Apesar de ser uma vulnerabilidade conhecida há muito tempo, o *buffer overflow* continua sendo almejado por atacantes devido à severidade das consequências que uma exploração bem sucedida pode produzir. Um explorador pode tomar o controle de um sistema inteiro caso encontre uma brecha em uma aplicação rodando em modo superusuário.

Para os administradores de sistemas, é importante que as aplicações sejam atualizadas constantemente, evitando a exposição do sistema a vulnerabilidades recém documentadas. Essa medida teria evitado muitas das infecções provocadas pelo worm *Code Red*, por exemplo.

Referências

[1] J. C. Foster, "Buffer overflow attacks : detect, exploit, prevent," 2005.

[2] I. Lopes, "O que é e como funciona o "buffer overflow"," may 2020.

[3] H. Tipton and M. Krause, *Information Security Management Handbook*. No. v. 1 in (ISC) 2 Press, CRC Press, 2007.

[4] 0x00SEC, "Examining the morris worm source code," may 2020.

[5] Microsoft, "Microsoft security bulletin ms01-033," june 2001.

[6] S. American, "Code red: Worm assault on the web," october 2002.