Introdução a exploração de Structured Excpetion Handlers

Autor: Brenno Rodrigues

Dezembro / 2015

Índice

Apresentação..3
1. Primeiramente, o que são Exceptions?.................................4
2. E sobre Structured Exception Handlers?...............................4
3. Impedindo a manipulação de EIP..7
4. Controlando EIP indiretamente..9
Bases para o texto...12

Apresentação

Em português é fácil encontrar textos sobre a escrita de exploits simples, mas percebi que faltava algo sobre exploits em cima de ***Structured Exception Handlers*.** Então, para não deixar você que ainda não aprendeu o english ficar perdido, resolvi escrever um pouco sobre o que aprendi com as variadas fontes que encontrei pela internet. Porém, irei supôr que você já saiba como funcionam esses exploits simples e ao menos tem uma ideia do que é o stack e como ele funciona, shellcode, um pouco de programação, etc. Se chegou por aqui sem fazer ideia sobre as coisas que acabei de falar, deixo este bom texto que achei pela internet e recomendo que comece por lá.

Antes de tudo, é bom também alertar que não sou algum tipo de profissional, professor, 1337 ou porra nenhuma. Só gosto de aprender sobre o que acho interessante e gosto de escrever, então só juntei as coisas e saiu o texto. No fim dele deixarei uns links (a maioria em inglês, sorry) que tive como base enquanto aprendia.

Let’s get it started.

1. Primeiramente, o que são Exceptions?

De acordo com o Wikipedia: “*são ocorrências de condições que alteram o fluxo normal da execução de programas de computadores.*” Ou melhor dizendo: algo fora do normal que exige um tratamento diferenciado.

Como programadores inteligentes cuidam dos seus softwares, é criado dentro do próprio programa meios para Manipular essas Exceções (**Exception Handlers**), que são as partes do software que serão chamadas quando algo sair diferente do que foi planejado (o sistema operacional, no caso o Windows, também cuida dessas tretas, mais na frente falo sobre).

2. E sobre Structured Exception Handlers?

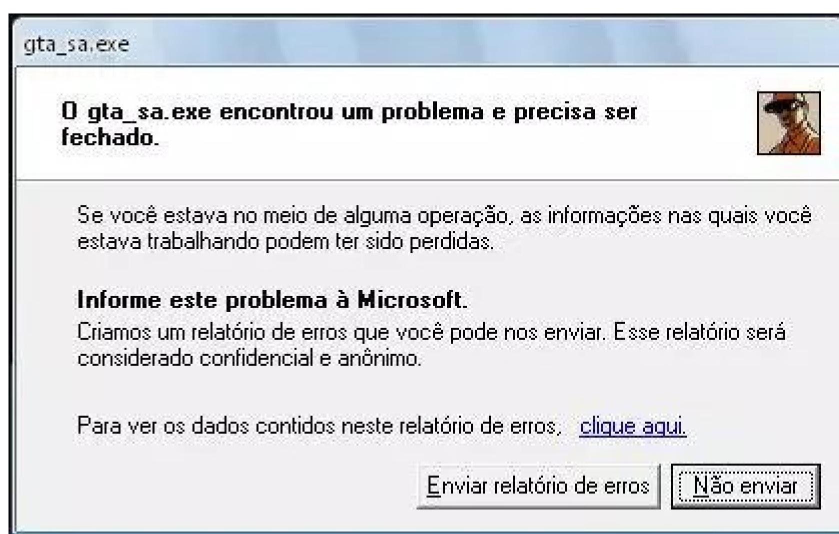
Structured Exception Handling (**SEH**), é o mecanismo nativo do Windows para Manipular Exceções.

O funcionamento é simples, observe essa bela implementação:

```
1  #include <string.h>
2  #include <windows.h>
3
4  int main (int argc, char **argv) {
5          char overflow_me[5];
6
7          __try {
8                  strcpy(overflow_me, argv[1]); // Bloco de código "Protegido"
9                  }
10         __except (GetExceptionCode()) {
11                         // Código que seria chamado no caso de um possível erro (exceção)
12                         }
13 }
```

Para melhores exemplos, dê uma sacada aqui no site da Microsoft.

O código acima mostra como um mecanismo para tratar exceções (**Exception Handler**) seria implementado pela aplicação (com certeza você já viu aquela janela que avisa que o programa teve um problema e precisa ser fechado, aquilo é trabalho do Exception Handler implementado pelo Windows, que é chamado quando o aplicativo ou não tem Exception Handler algum, ou nenhum deles conseguiu lidar com aquela exceção especifica).



Olha a janelinha do Windows aê

Dependendo das funções do programa, normalmente, se tem muito mais que um Exception Handler, afinal, elas (as Exceptions) podem ser dos mais variados tipos (divisão por zero, buffer overflow...).

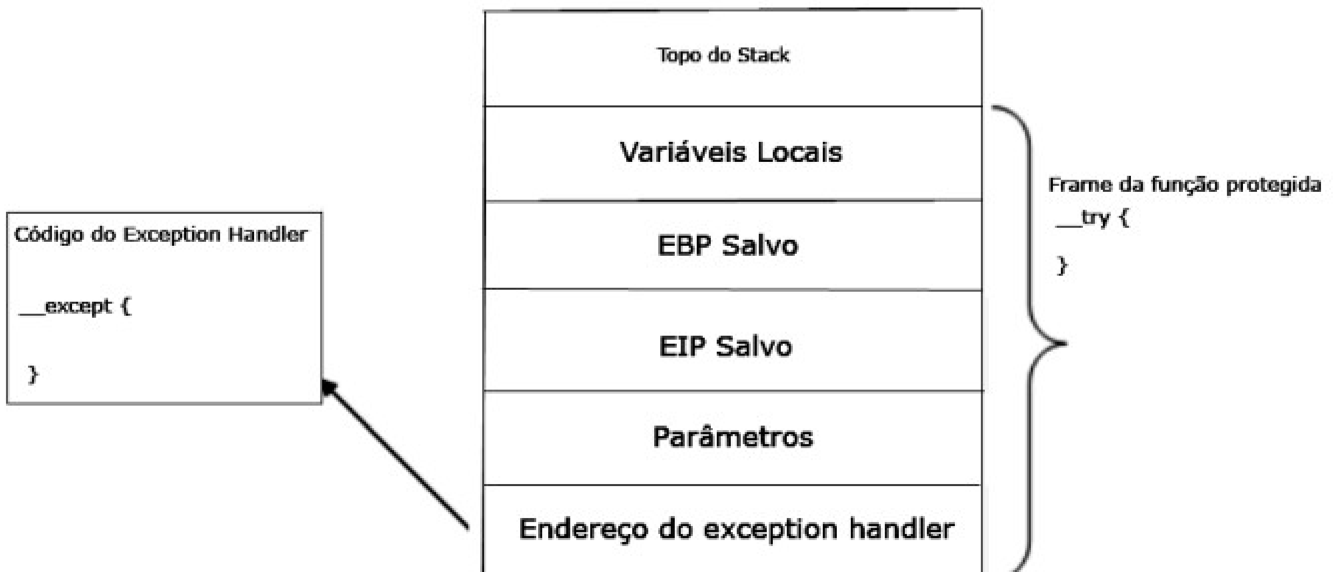
Ok, agora podemos dizer que sabemos o básico do que estamos falando. Vamos então seguir para a parte security da coisa.

Se você seguiu meu conselho, deve saber como um exploit simples para falhas do tipo stack overflow funciona. Sabe que toda a jogada está em sobreescrever o ponteiro EIP com um endereço que está sob seu controle, para assim redirecionar o fluxo do programa para que ele faça o que você quiser, coisa não muito difícil de se

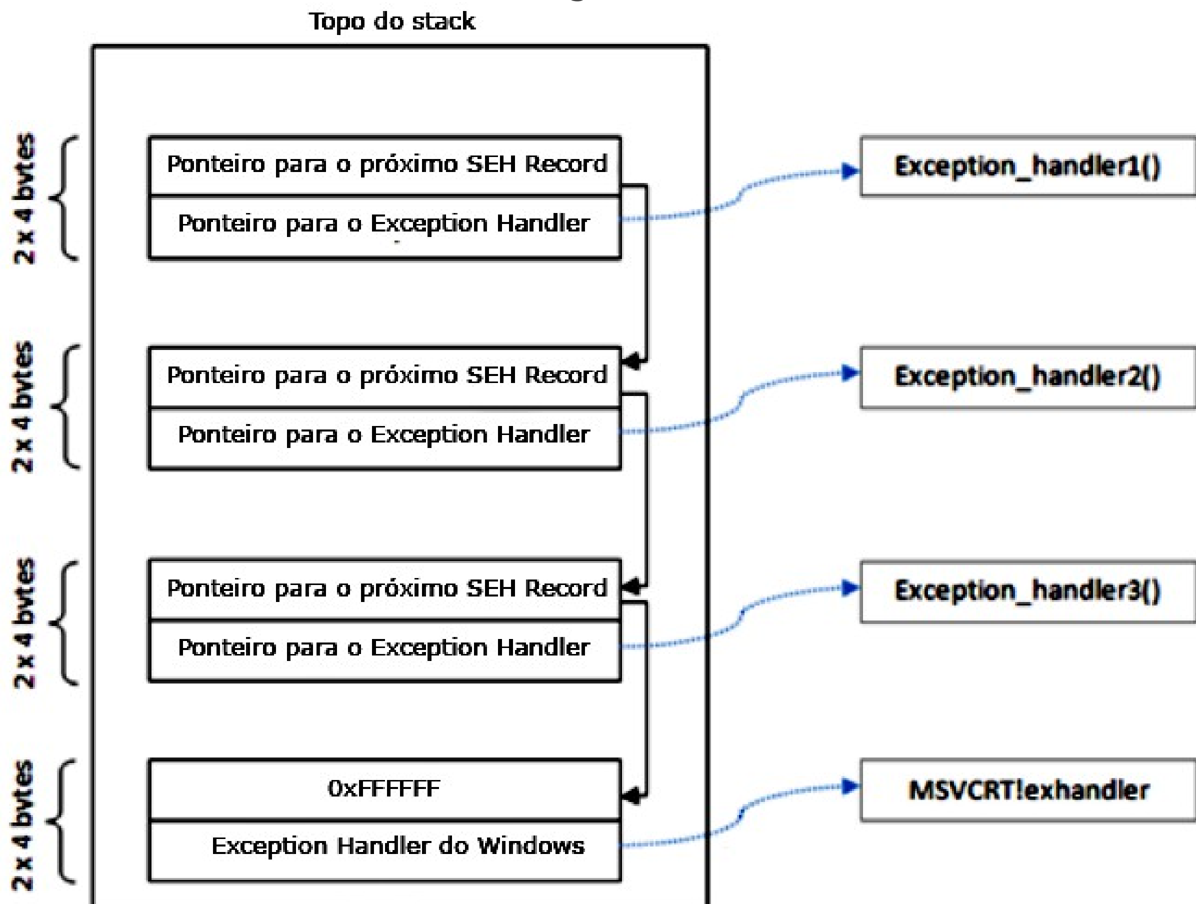
fazer se o ambiente for propício.

Vamos então descobrir como os Structured Exception Handlers podem atrapalhar essa jogada e como se supera essa forma de defesa.

3. Impedindo a manipulação de EIP



Nessa imagem podemos ver a organização do stack quando uma função "protegida" é chamada. Observe onde os Exception Handlers se localizam. Eles são organizados dessa forma:



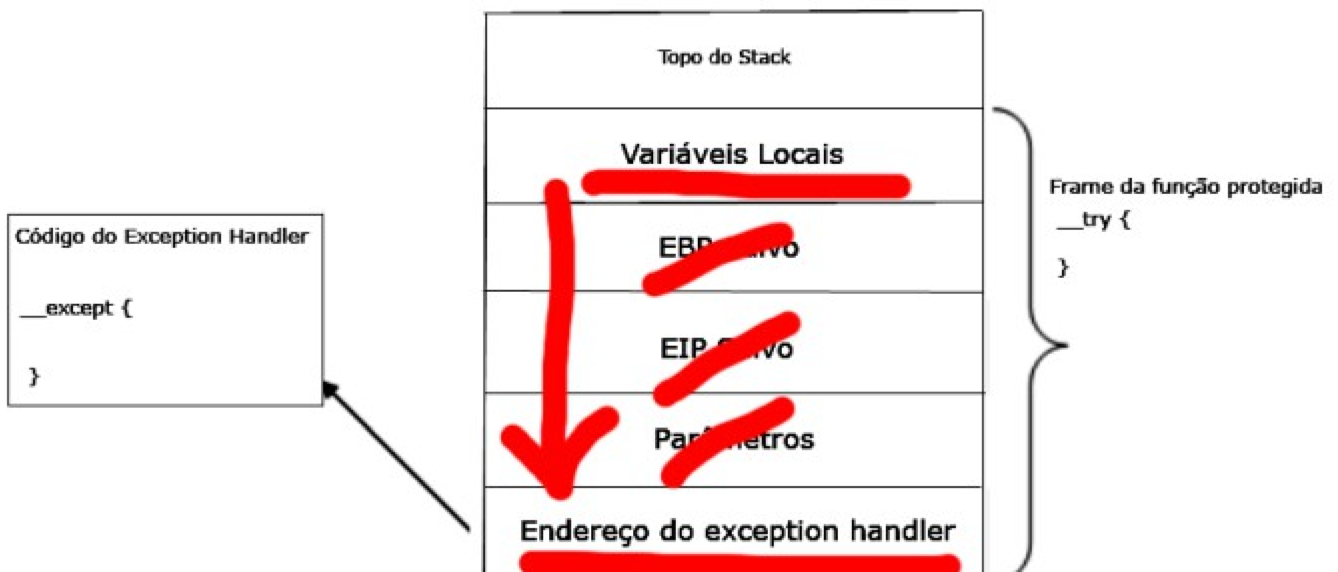
Cada pedacinho desses (os **SEH Record**), é formado por dois elementos de 4 bytes cada. O primeiro aponta para o próximo SEH Record (no caso do atual SEH Record não conseguir tratar da exceção), e o segundo para o endereço do código responsável por tratar a exceção. Todos, organizados um abaixo do outro, formam uma “sequência” ou “cadeia” de métodos para o tratamento de exceções, por isso é chamada de “**SEH Chain**“.

E, no caso de algo dar errado, como o programa sabe onde fica a SEH Chain?

Seu endereço fica no topo do Win 32 Thread Information Block (**FS:[0x00]**), que é uma estrutura de dados criada pelo Windows para guardar informações sobre uma Thread.

Após encontrar uma falha do tipo buffer overflow, você vai seco com o objetivo de atropelar tudo e sobreescrever EIP, mas se você se deparar com SEHs, vai poder perceber que, após o buffer ser estourado e sair sobreescrevendo dados, EIP, ao invés de ter sido corrompido pelo seus dados incluidos maliciosamente, irá apontar direto para o topo da SEH Chain e não será tão fácil como antes para explorar a falha.

4. Controlando EIP indiretamente



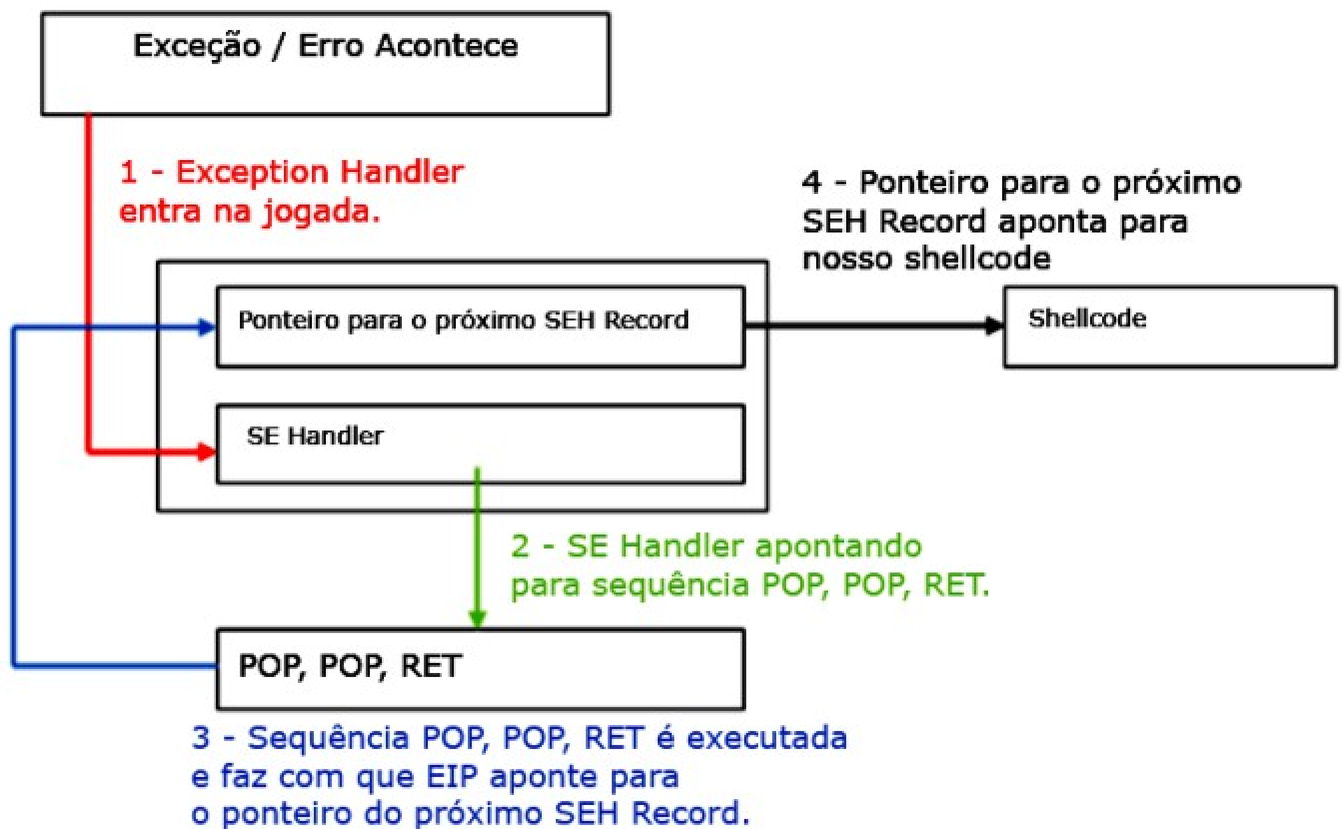
Corrompendo tudo ladeira abaixo.

Ao observar a imagem onde nos é mostrado em que lugar do stack o ínicio da SEH Chain é armazenado, veremos a proximidade que ela fica do buffer que exploramos. Se enfiarmos mais e mais dados, assim como EIP seria em um ataque básico, podemos alter o SEH Chain. E o que fazer com esse poder?

Fácil. Enganar o sistema operacional e fazer ele rodar nosso shellcode.

Como sabemos, o SEH Record tem oito bytes (quatro bytes para o endereço do próximo SEH Record, e os outro quatro apontam para a função que tratará a exceção, pode observar na imagem2). Queremos que os primeiros quatro bytes apontem para onde nosso shellcode está, e que os outros quatro apontem para esses quatro bytes anteriores, para que assim nosso shellcode seja executado. Para conseguir fazer isso há diferentes métodos, o mais comum é usar a sequência de instruções *POP, POP, RET*.

Iremos procurar um endereço que aponte para o ínicio dessa sequência de instruções e usa-lo para sobreescrever os últimos quatro bytes do SEH Record.



O exploit ficaria assim: *[lixo] [jmp shellcode] [pop pop ret] [shellcode]*

Claro que nem tudo são flores. Por exemplo, a partir do Windows XP SP 1, quando ocorre alguma exceção e o Strucured Expection Handling entra em ação, todos os registros são zerados, o que impede que se faça o uso deles para apontar para nosso shellcode. Isso dificulta, mas não impede a exploração, tanto que mostrei como funciona o exploit que não depende dos registros (por isso que usamos *pop pop ret*).

Outro exemplo que pode dificultar e as vezes impedir um exploit (ou impedir que o mesmo seja multi-plataforma) é o **SafeSEH** implementado a partir do Windows XP SP2 e Windows Server 2003. Quando um módulo do executável for compilado com

essa opção ativa, apenas endereços registrados na lista de SEH podem ser usados como Handlers. Ou seja, nada de usar um endereço qualquer para indicar a *POP, POP, RET* (a não ser que ele esteja na lista). Há algumas formas de burlar esse artificio, mas assim como DEP (Data Execution Prevention) e Stack Canaries, deixo isso para outro dia (ou seja, quando estudar mais fundo sobre).

Bases para o texto

- https://isisblogs.poly.edu/2011/11/23/seh-record-exploitation/
- https://www.corelan.be/index.php/2009/07/25/writing-buffer-overflow-exploits-a-quick-and-basic-tutorial-part-3-seh/ <- de onde copiei boa parte das imagens aqui usadas.
- https://msdn.microsoft.com/en-us/library/windows/desktop/ms680657%28v=vs.85%29.aspx
- https://pt.wikipedia.org/wiki/Tratamento_de_exce%C3%A7%C3%A3o
- https://en.wikipedia.org/wiki/Microsoft-specific_exception_handling_mechanisms
- https://en.wikipedia.org/wiki/Memory_safety
- https://en.wikipedia.org/wiki/Win32_Thread_Information_Block+
- https://pt.wikipedia.org/wiki/Thread_%28ci%C3%AAncia_da_computa%C3%A7%C3%A3o%29