Grupo de Resposta a Incidentes de Segurança – GRIS

Departamento de Ciência da Computação

Universidade Federal do Rio de Janeiro

Buffer Overflow – Uma introdução teórica

Por: Raphael Duarte Paiva

Universidade Federal do Rio de Janeiro
Instituto de Matemática
Departamento de Ciência da Computação
Grupo de Resposta a Incidentes de Segurança

Rio de Janeiro, RJ – Brasil

Título

GRIS-2008-A-001

Raphael Duarte Paiva

A versão mais recente deste documento pode ser obtida na página oficial do GRIS

Uma palavra do autor

Este artigo visa introduzir os leitores à um tipo de vulnerabilidade conhecida como Buffer Overflow, ou Buffer Underrun, entre outros nomes. Partindo dos conceitos iniciais, mesmo uma pessoa que nunca ouviu falar de tal termo poderá facilmente ler este documento.

O objetivo do autor é apresentar, para quem nunca conheceu, explicar o funcionamento com os detalhes pertinentes desde o nível de abstração do hardware, com uma linguagem simples, objetivando o entendimento e aprofundamento para usuários tanto de nível iniciante como os de nível mais avançado.

No texto, serão tratados tópicos que explicarão o básico para o simples entendimento do conceito de Buffer Overflow, caminhando para uma apresentação dos tipos mais famosos de ataques e suas idéias básicas.

Textos futuros tratarão das defesas contra este tipo de vulnerabilidade e sua aplicação prática com exemplos de códigos e uma análise do primeiro exploit baseado em buffer overflow documentado: o Morris Worm.

Indice:

1 – Os conceitos básicos

1.1 – Um breve histórico

Buffer Overflows foram entendidos por volta de 1972, quando o Computer Security Technology Planning Study (documento de planejamento de estudos da USAF) definiu a técnica como o seguinte:

"Um código realizando tal função não checa os endereços de fonte e destino apropriadamente, permitindo que partes do ***monitor*** *sejam superpostas pelo usuário. Isso pode ser usado para injetar códigos no* ***monitor*** *que irão permitir o usuário a tomar o controle da máquina." (*Página 61 http://csrc.nist.gov/publications/history/ande72.pdf).

Hoje em dia, o **monitor**, poderia ser referido como o kernel.

A documentação mais antiga de Buffer Overflow data de 1988. Foi uma das muitas vulnerabilidades exploradas pelo Morris worm para se propagar pela internet. O programa-alvo foi um serviço Unix, chamado finger.

Em 1995, Thomas Lopatic redescobriu sozinho o buffer overflow e publicou seus achados na lista de emails da Bugtraq. Um ano depois, Elias Levy (Também conhecido como Aleph One), publicou na revista Phrack, o artigo "Smashing the Stack for Fun and Profit" (Arrasando a Pilha por Diversão e Lucro), uma introdução passo-a-passo para realizar um stack-based buffer overflow.

Desde então, pelo menos 2 worms exploraram buffer overflows para comprometer um número grande de sistemas. Em 2001, o Code Red Worm explorou um buffer overflow no ISS (Internet Information Services) 5.0 da Microsoft e em 2003, o worm SQL Slammer colocou em risco maquinas rodando o Microsoft SQL Server 2000.

Ainda em 2003, buffer overflows em jogos licenciados para o vídeo-game Xbox da Microsoft foram explorados para permitir que softwares não licenciados rodassem no console sem a necessidade de modificações no hardware, conhecidas por modchips. O PS2 Independence Exploit também usou um buffer overflow para conseguir o mesmo efeito para o video-game Playstation 2 da Sony.

1.2 – Buffer - o que é?

Em computação, um buffer é uma região temporária da memória onde são guardados dados para serem transportados de um lugar para o outro.

Uma aplicação freqüente de um buffer é quando dados são capturados de um dispositivo de entrada (um teclado ou mouse, por exemplo) e enviados à um dispositivo de saída(monitor, impressora).

Exemplo:

Digamos que o usuário está em uma aplicação de terminal, onde tudo o que ele digita, é impresso na tela.

O processo de imprimir o que o usuário digita, simplificadamente, segue um padrão parecido com o seguinte:

1. O Usuário aperta uma tecla.
2. Um buffer préviamente programado para tal fim, armazena o dado vindo da entrada padrão (o teclado).
3.
4. O dado contido no buffer é então copiado para uma variável interna do sistema, em seguida, processado para se tornar legível à saída padrão (geralmente usa-se um padrão de codificação de caracteres, como a tabela ASCII, por exemplo).

5. O dado é então, copiado para a saída padrão e, conseqüentemente, impresso na tela do terminal.

Buffers também são utilizados em aplicações onde a taxa de dados recebidos é maior do que a taxa em que os dados são processados, como por exemplo em uma fila de atendimento: Enquanto um banco, por exemplo, tem 5 terminais de atendimento, pessoas são atendidas em grupos de 5. Caso o número de pessoas a serem atendidas seja maior que cinco, estas que chegaram por último formam uma fila enquanto esperam o termino de um atendimento para serem atendidas. A fila funciona como um buffer, armazenando as pessoas em um lugar próximo de seu destino, enquanto

ainda não podem se locomover até ele.

Em um exemplo muito mais simples, imagine-se em uma sala de aula e que você deve copiar o conteúdo do quadro para seu caderno. O procedimento para a maioria das pessoas seria olhar o quadro, memorizar uma parte do conteúdo e escrevê-la em seu caderno. De fato, sua memória serviu como um buffer entre a entrada de informações (o quadro) e a saída (seu caderno).

Buffers são utilizados tanto em Hardware quanto em Software, porém muito mais utilizados neste ultimo.

1.3 – Overflow

O conceito básico de um overflow é um transbordamento, ou seja, um overflow ocorre quando o volume de determinado objeto ou substância é maior do que a capacidade de seu compartimento.

Um rio raso exposto à chuvas fortes aumenta consideravelmente seu volume de água, podendo levar o nível de água a ser maior que a altura de suas margens, fazendo-o transbordar. Este fato caracteriza um overflow (transbordamento), o que leva a água do rio se espalhar por locais onde a mesma não deveria estar naturalmente.

Agora que temos consolidados os conceitos de buffer e de overflow, podemos juntá-los para conhecer o conceito de buffer overflow.

1.4 – Buffer Overflow - Uma visão geral

O buffer overflow é um erro de programação que pode resultar em execução errônea de um programa, acesso indevido à áreas de memória, terminação do programa ou uma possível falha de segurança em um sistema.

Tecnicamente falando, um buffer overflow consiste em um processo de armazenar, em um buffer de tamanho fixo, dados maiores que o seu tamanho.

Exemplo:

Imagine que um programa declara duas estruturas de dados adjacentes na memória do computador: um vetor de caracteres de 8 bytes, A, e um inteiro de 2 bytes, B, enche A de zeros e B recebe o número 9, como no seguinte esquema:

A	A	A	A	A	A	A	A	B	B
0	0	0	0	0	0	0	0	0	9

Agora imagine o que aconteceria se o programa nos pedisse uma palavra e tentássemos inserir a palavra “excessivo” dentro de A.

A palavra excessivo tem 9 letras, conseqüentemente, 10 caracteres, pois um caractere nulo é concatenado ao final para indicar o fim de uma string (vetor de caracteres). Separando a palavra em caracteres seria algo como: 'e', 'x', 'c', 'e', 's', 's', 'i', 'v', 'o','\0', sendo o '\0' o caractere nulo, que consiste em um um byte zerado.

Sabendo que cada caractere ocupa um byte, a string A, então, tem apenas 8 bytes, porém estamos inserindo 10 bytes, que é um tamanho maior do que A comporta.

Como no exemplo do rio transbordando, A vai transbordar, caracterizando um *overflow*.

Quando ocorre um *overflow*, áreas da memória adjacentes à estrutura de dados a ser preenchida serão ocupados com os dados excedentes. Neste caso, os 8 bytes de A serão ocupados, e haverão 2 bytes excedentes que serão alocados nos 2 bytes adjacentes à A, que é onde reside B, portanto, o valor de B será sobrescrito com os 2 bytes que A não suportou, como no seguinte esquema:

A	A	A	A	A	A	A	A	B	B
'e'	'x'	'c'	'e'	's'	's'	'i'	'v'	'o'	'\0'

B é um inteiro que antes valia 9. Agora, seus 2 bytes estão preenchidos com os caracteres 'o' e '\0'. Em sistemas Little-Endian (sistemas em que o byte menos significativo é o da esquerda), o valor dos bytes de B seriam os correspondentes binários à tabela ASCII do caractere nulo (00000000) e do caractere 'o' (01101111). Concatenado-os, temos o valor binário de B: 0000000001101111 que equivale ao decimal 111.

Como o programa em questão tem apenas 10 bytes reservados na memória, se tentássemos escrever uma palavra maior ainda, os excedentes teriam de ser alocados em áreas de memória não

reservadas ao programa, resultando numa falha de segmentação e, conseqüentemente, a terminação do programa.

As conseqüências do tipo de erro citado no exemplo acima, podem resultar em um funcionamento errôneo do programa, pois imagine que B fosse uma variável em que seria armazenada a idade do usuário e em seguida feito algum processamento em cima do dado. Uma idade de 111 anos certamente não levaria à um resultado esperado. Logicamente, este é um exemplo simples de um buffer overflow. Erros deste tipo em projetos maiores podem levar à bugs desastrosos, horas de trabalho perdidas para depuração, além de falhas de segurança que levarão ao surgimento de exploits podendo causar sérios transtornos à todos que usam o programa.

1.5 – Exploits e tipos de ataques baseados em Buffer Overflow

Exploits são pedaços de código, seqüências de comandos, ou até mesmo programas que se aproveitam de vulnerabilidades ou bugs em softwares, visando causar comportamento errôneo, inesperado ou até mesmo ganhar acesso privilegiado à um sistema ou causar ataques de negação de serviço (DoS).

A maioria dos Exploits visa ganhar acesso de nível de superusuário à sistemas ou derrubar os mesmos. Pode-se fazer uso de vários exploits diferentes para galgar níveis de acesso até chegar ao superusuário.

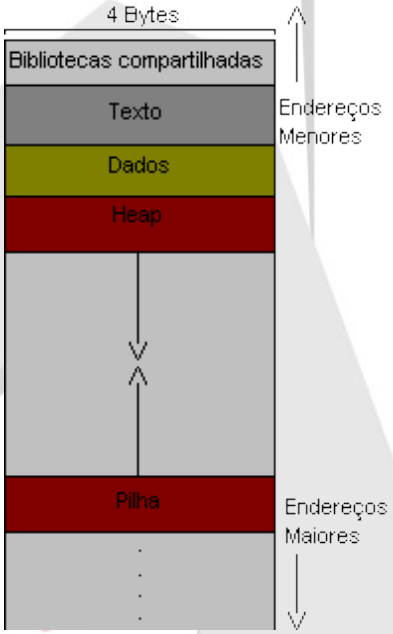
Os tipos mais comuns de ataques baseados em buffer overflow são:

- Stack-based buffer overflow
- Heap-based buffer overflow
- Return-to-libc attack

2 – Dissecando

2.1 – A estrutura de um programa na memória

Qualquer programa compilado (e isto inclui interpretadores de linguagens) possui uma forma estruturada em código de máquina denotando uma seqüência de instruções que serão executadas pelo processador. Tal seqüência é carregada na memória quando um programa é chamado à execução.



A estrutura de um programa na memória se divide em quatro partes: texto, dados, stack (pilha) e heap.

A região do texto contém as instruções do programa propriamente ditas e dados somente-leitura. Essa região é geralmente somente-leitura, para evitar códigos mutáveis. Uma tentativa de escrita nessa área resulta em uma falha de segmentação.

A região dos dados contém todas as variáveis globais e estáticas do programa.

A região de heap serve para a alocação dinâmica de variáveis, através de instruções do tipo malloc();

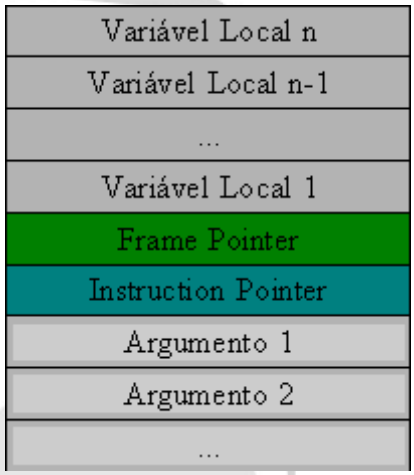
A pilha é a região mais interessante para os fins deste texto. Trata-se de um bloco de memória onde são armazenadas informações vitais para subrotinas: variáveis locais, parâmetros e os valores de retorno.

2.2 – Conhecendo melhor a pilha

O conceito de pilha é bastante simples. Uma pilha adota um padrão conhecido como LIFO (Last In First Out – último a entrar, primeiro a sair).

Imagine uma pilha de pratos que devem ser lavados. A pessoa que lavará os pratos, começará pelo topo da pilha, que foi o ultimo prato a ser colocado na mesma, porém, será o

primeiro a ser lavado. Em termos de estruturas de dados, aplica-se o mesmo conceito. O último dado a entrar, será o primeiro a ser tratado em estruturas organizadas por pilhas (stacks).



Na memória, a pilha consiste de frames que são colocados no topo quando uma subrotina é chamada e retirada da pilha no termino da subrotina.

Um frame geralmente guarda os seguintes dados sobre a subrotina em ordem da base para o topo:

- Espaço para os parâmetros da subrotina;
- O endereço de retorno da subrotina;
- Um valor chamado comumente de Frame Pointer;
- Espaço para variáveis locais.

Definindo o escopo à arquiteturas Intel x86 de 32 bits, quando temos chamadas à subrotinas (funções) em programas, o processador recebe uma instrução chamada *CALL*. Esta instrução armazena no frame da respectiva subrotina duas informações vitais para o processador: o IP (Instruction Pointer) e o FP (Frame Pointer), que indicam respectivamente, a posição de memória para onde o processador deve continuar lendo após o termino da subrotina e um valor arbitrário para facilitar o trabalho do processador na hora de ler e escrever em variáveis locais e ler parâmetros.

O FP ajuda o processador de modo que endereços lidos dentro do frame que possuam um valor maior que o endereço do FP é com certeza um parâmetro e endereços lidos dentro do frame que possuam um valor menor que o endereço do FP é uma variável local.

A soma do espaço reservado para o buffer desprotegido com o espaço ocupado pelo FP é chamado de *payload space*.

2.3 – Buffer Overflow - uma visão mais detalhada

Como vimos, podemos utilizar o buffer overflow para reescrever o IP. Ora, se o IP está logo após o espaço para as variáveis locais e o FP, o que aconteceria se ultrapassássemos este espaço?

Assim como no primeiro exemplo de buffer overflow, imagine que temos um programa que chama uma subrotina para pedir uma string ao usuário. Se soubermos o tamanho máximo da string e esta fosse a única variável da subrotina, poderíamos ultrapassá-lo e reescrever o valor do IP.

Digamos que o programa em questão peça o nome de um arquivo em um sistema Unix. Sabendo que o tamanho máximo do nome de um arquivo em tal sistema é de 256 caracteres, fica fácil arquitetar a reescrita do IP, considerando que o programa não faz verificação de tamanho de string e que o está rodando num sistema desprotegido, basta escrever 264 caracteres para que o IP receba os últimos 4 bytes da entrada, ou seja, os quatro últimos caracteres, pois o IP guarda um inteiro de 4 bytes e cada caractere tem 1 byte (utilizando o sistema ASCII), ora, tenho 8 caracteres excedentes de 1 byte cada, então o IP terá o valor igual à concatenação dos valores binários dos 4 primeiros caracteres excedentes e o IP dos 4 últimos.

2.4 – Técnicas e ferramentas de Buffer Overflow

Agora que sabemos como alterar o IP, como podemos utilizar isso para fins maliciosos? Três técnicas se destacam na exploração do buffer overflow. Esta é uma ferramenta importante – o shellcode – serão explicadas a seguir.

2.4.1 – Stack-based Buffer Overflow

Como dito em 2.1, o processador lê uma série de instruções. Tais que são referenciadas por bytes. Com este conhecimento, podemos então, no exemplo, ao invés de escrevermos um nome, poderíamos enviar os caracteres referentes aos bytes de instruções do processador e reescrever o IP como o endereço referente ao início dos bytes que acabamos de escrever.

Deste modo, as instruções referentes aos bytes que escrevemos serão executadas, pois o IP aponta para o início dos bytes maliciosamente escritos, então, quando a subrotina terminar, o processador lerá a instrução apontada pelo IP, que será o primeiro byte que acabamos de escrever.

Isto caracteriza uma injeção de código por buffer overflow, diretamente na pilha. É o tipo de

ataque baseado em buffer overflow mais comum e simples. O código inserido é normalmente chamado de *shellcode*.

2.4.2 – A estrutura dos Shellcodes

Shellcodes são pedaços de código legíveis ao processador, ou seja, bytes referentes às instruções. Como geralmente shellcodes são passados como strings, representamos cada byte por "\ xNN", onde "\" significa que estamos passando um byte (não um caractere), "x" significa que seu valor está na base hexadecimal e NN é o valor do byte na base hexadecimal. Vale lembrar que um byte nulo "\x00", ou simplesmente "\0" é o caractere terminador de string, então, como o shellcode consiste de uma string composta por bytes, não podemos ter bytes nulos, pois funções que trabalham com strings interpretam um byte nulo como o fim da mesma e terminam o seu processamento. Obviamente, queremos que nossa string maliciosa seja processada inteira, portanto, um byte nulo é um problema. Além do byte nulo, alguns outros bytes como linefeed("\n", ou "\x0A"), carriage return("\x0D"), etc. podem nos causar problemas, como quebrar o shellcode em dois ou mais (linefeed), como se entrássemos com 2 linhas separadas de shellcode.

Podemos resolver estes problemas de vários modos, dependendo da instrução que desejamos representar. Torna-se óbvia a necessidade do conhecimento da linguagem de máquina neste caso.

No caso de querermos construir um shellcode que imite uma função, como System(), que executa o comando de sistema passado por argumento, devemos seguir os seguintes passos:

- Construir um programa que simplesmente chame a função System(), com o argumento contendo o comando de sistema que desejamos executar.
- Analisar o seu código assembly do programa por meio de um debugger, como o gdb, por exemplo, conseguindo assim acesso ao código assembly da função System.
- Pelo próprio gdb, podemos ter acesso aos bytes referentes a cada instrução pelo comando "x/bx <nome_da_função>"

Com os bytes em mão, podemos então construir o shellcode, concatenado-os em uma string, tomando os cuidados e adaptações necessárias para o shellcode não conter bytes especiais.

Assim, com o shellcode pronto, devemos preparar o buffer malicioso para ser construído.

Este deve conter:

- Dados para encher o buffer até o início do shellcode
- O shellcode em si.
- O valor para reescrever o IP com o endereço do início do shellcode

Um problema comum de ataques por buffer overflow baseados na pilha é que não sabemos exatamente onde é o início do payload space devido à aleatoriedade dos dados na memória, devido ao fato então é comum termos shellcodes bem menores que o payload space, e, precedidos de bytes NOP ("\x90", No Operation), os quais o processador ignora e pula para a próxima instrução. Deste modo, temos um espaço maior para erros na hora de presumir o endereço de retorno. Caso o shellcode não comece no endereço escolhido, o processador encontrará NOPS até o início do shellcode.

2.4.3 – Conhecendo melhor a heap

Geralmente, a memória dinâmica é alocada de uma grande área de memória livre, chamada de heap, ou free store. A localização de dados na heap não é prevista pelo programa, então dados na heap costumam ser acessados indiretamente, por meio de referências. Quando usamos uma função da família malloc(), estamos alocando espaço para uma estrutura de dados na heap e retornando o endereço de memória em que ela foi alocada. Este endereço é normalmente armazenado em uma variável que passa a ser a referência àquela estrutura na heap, então qualquer operação de leitura e escrita é feita através da referência, nunca diretamente.

O sistema operacional geralmente dá ao programa, quando este é carregado na memória, uma heap de um tamanho específico, porém este espaço pode ser modificado em tempo de execução. Geralmente, neste processo, o kernel divide a heap em setores menores pela memória, resultando em fragmentação e abstraindo esta tarefa do programador, que por sua vez, não precisa se preocupar com o tamanho do heap.

Arquivos executáveis tanto PE ou PE32 (Windows), quanto ELF ou ELF32 (Linux) possuem o que chamamos de "optional header" que, como o nome diz, é um cabeçalho opcional que contém informações sobre o quão grande o espaço na heap é esperado, para uma alocação mais precisa quando o programa é carregado.

É importante ressaltar que o heap cresce na direção oposta da pilha, ou seja, seu "topo"

cresce em direção aos endereços maiores da memória, enquanto a pilha cresce em direção aos endereços menores da memória.

Além do espaço reservado para as estruturas de dados do programa, um bloco no heap também guarda valores usados para a sua administração, como tamanho do bloco, espaço utilizado, próximo bloco e bloco anterior. Os valores armazenados são dependentes de implementação. Todo run-time environnement possui o seu próprio modo de administrar a heap, porém, um programa pode definir o seu próprio modo de administração da heap.

Como o espaço na heap é variável para um programa em execução, não há como definir limites para este espaço. Então, nem o kernel, nem o processador são capazes de detectar um overflow na heap.

2.4.4 – Heap-based Buffer Overflow

Este é um tipo muito mais difícil de ser explorado devido ao fato do alvo ser a heap. Entre as dificuldades estão:

- Na heap, não existem valores os quais o processador usa para saber qual a próxima operação a executar, apenas valores de organização do próprio heap.
- Devido ao fato do heap crescer em direção aos valores maiores da memória e podermos apenas escrever nesta mesma direção, não é possível reescrever os valores de controle do bloco atual, apenas dos blocos posteriores.
- Como a organização da heap pode variar, o atacante precisa saber qual é a implementação de heap do programa-alvo.

Para explorar um buffer overflow na heap é necessário encontrar uma vulnerabilidade em alguma função interna de administração da heap, de modo que esta processe o buffer alvo na heap. Deste modo:

- Precisamos explorar a heap para sobrescrever os valores de administração de um bloco alvo e assim inserir o shellcode, construindo um bloco malicioso na heap.
- Os valores de administração devem ser alterados no bloco de forma que este, ao ser analisado por uma função, como free(), por exemplo, seja processado, diretamente ou por meio de outra função, como a unlink(), que é chamada dentro da free(), por exemplo.
- Além de modificados para ativar uma função, os valores de administração devem ser

reescritos para causar um overflow no frame da função que processa o bloco, para direcionar a execução do programa para o shellcode inserido.

Outro modo de explorar o heap overflow é estourar o heap até chegar na pilha e então sobrescrever o IP do frame do topo da pilha.

Neste ponto, uma questão importante é conveniente: Por que um shellcode no heap se precisamos explorar um overflow na pilha também?

A resposta pode variar, porém a mais razoável é a questão do espaço. Funções podem ser vulneráveis e mesmo assim ter uma quantidade tão pequena de buffers que não suportaria um shellcode. É razoável, neste caso, inserir o shellcode na heap.

2.4.5 – Return-to-libc Attack

Uma das defesas para um ataque de buffer overflow baseado em pilha é não dar permissão de execução na pilha. Deste modo, não é possível tentar executar um shellcode na pilha. Existe uma alternativa, que é explorar um código que já existe nas rotinas do programa. Um código que está em quase qualquer coisa(incluindo o sistema operacional), são os códigos da libc, a biblioteca padrão do C. Além de estarem presentes na maioria dos sistemas, eles possuem uma posição estática na memória em um sistema.

O método para pular para uma função da libc é o mesmo do buffer overflow baseado em pilha: reescrever o IP com o endereço da função desejada. O endereço de uma função da libc pode ser facilmente descoberto, em um ambiente sem proteção de aleatoriedade nos espaços de endereço, usando um programa de teste que imprima o endereço da função, por exemplo.

Devemos, então, reescrever o IP com o endereço da função desejada, seu endereço de retorno e então seus argumentos.

Referências

Smashing The Stack for Fun and Profit - http://insecure.org/stf/smashstack.html
Wikipedia em inglês nos seguintes tópicos:

- Buffer Overflow - http://en.wikipedia.org/wiki/Buffer_overflow

- Stack buffer overflow - http://en.wikipedia.org/wiki/Stack_buffer_overflow
- Heap Overflow - http://en.wikipedia.org/wiki/Heap_overflow
- Return-to-libc attack - http://en.wikipedia.org/wiki/Return-to-libc_attack
- Shellcode - http://en.wikipedia.org/wiki/Shellcode

A Heap of Risk - http://www.heise-online.co.uk/security/A-Heap-of-Risk--/features/74634

w00w00 on Heap Overflows - http://www.w00w00.org/files/articles/heaptut.txt

Arc Injection / Return-to-libc - http://www.acm.uiuc.edu/sigmil/talks/general_exploitation/arc_injection/arc_injection.html

Agradecimentos

O primeiro e mais óbvio agradecimento vai para minha família, sem a qual não chegaria onde estou hoje e certamente não estaria realizando este artigo.

Gostaria de agradecer também a todo a equipe GRIS, pelo apoio, camaradagem, ensinamentos, trabalho, compromisso e as muitas risadas que compartilhamos quando estamos juntos.

Outro agradecimento vai para toda a comunidade online que compartilha conhecimento em ótimas fontes de sabedoria.

Finalmente, um agradecimento especial para o amigo e fundador do GRIS, Breno G. de Oliveira, por sua disposição em compartilhar seu enorme conhecimento a qualquer hora e sempre com uma vontade animadora de ajudar.