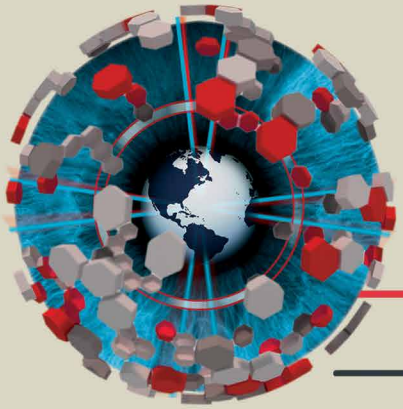


H2HC

HACKERS TO HACKERS CONFERENCE

MAGAZINE

ENGENHARIA REVERSA DE SOFTWARE

A NOVA COLUNA POR FERNANDO MERCÊS!

STACK FRAMES O QUE SÃO E PARA QUE SERVEM? PARTE II
POR YGOR PARREIRA E FILIPE BALESTRA

A FALHA HEARTBLEED E SEUS IMPACTOS NOS NEGÓCIOS
POR JULIO MOREIRA

NOVIDADE: COLUNA CURIOSIDADES

H2HC MAGAZINE
EDIÇÃO 8
AGOSTO DE 2014

Direção Geral
Rodrigo Rubira Branco
Filipe Balestra

Direção de Arte / Criação
Amanda Vieira

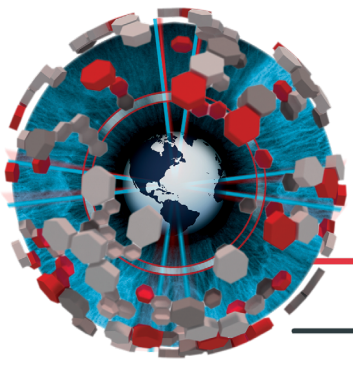
Coordenação Administrativa / Mídias Sociais
Laila Duelle

Redação / Revisão Técnica
Gabriel Negreira Barbosa
Ygor da Rocha Parreira
Jordan Bonagura
Leandro Bennaton

Agradecimentos

Equipe do **Renegados Cast**
Julio Moreira
Raghudeep Kannavara
Fernando Mercês

Índice



H2HC

HACKERS TO HACKERS CONFERENCE

MAGAZINE

A falha *Heartbleed* e seus Impactos nos Negócios

POR JULIO MOREIRA

O mundo da tecnologia (em especial o da segurança da informação) ficou em polvorosa quando, em abril/2014, foi divulgada uma falha crítica na biblioteca de criptografia OpenSSL[1], que ficou conhecida como *HeartBleed* [2][3]. A discussão que se viu em fóruns e na mídia, especializada ou não, colocava em dúvida qual o real impacto desta falha: seria possível invadir o site? Interceptar comunicação criptografada? Qual a abrangência do impacto causado? Apenas os sites que usam SSL? E os demais serviços (e-mail, webmail, VPN), estariam seguros? Quais os ambientes afetados (Windows, *NIX)?

Como diria Jack, o Estripador, vamos por partes: antes de qualquer coisa é importante entender, ainda que superficialmente, o que é o OpenSSL e quais são as suas aplicações. Uma vez esclarecida sua utilidade, devemos então entender qual foi a falha descoberta, como ela pode ser explorada e o que ela nos dá de informação do ambiente comprometido para, por fim, estabelecer uma estratégia de correção, com um plano de comunicação estabelecido (e adequado aos diversos níveis hierárquicos da empresa), a fim de não causar um temor desnecessário tampouco ignorar os problemas futuros que possam ocorrer.

Começando pelo começo: o que é o tal do OpenSSL ?

O OpenSSL é um projeto colaborativo e de código aberto que desenvolve e mantém um conjunto de ferramentas para implementação dos protocolos Secure Socket Layer (SSL v2/3) e Transport Layer Security (TLS v1), bem como uma biblioteca robusta de criptografia para propósitos gerais [4].

De acordo com a wikipedia[5], "O Transport Layer Security - TLS (em português: Segurança da Camada de Transporte) e o seu antecessor, Secure Sockets Layer - SSL (em português: Protocolo de Camada de Sockets Segura), são protocolos criptográficos que fornecem segurança de comunicação na Internet para serviços como email (SMTP), navegação por páginas (HTTPS) e outros tipos de transferência de dados." As informações sensíveis trafegadas variam de acordo com a aplicação executada no servidor em questão, mas, de uma forma geral, é possível citar informações de autenticação (usuário e senha), transações bancárias, e-mails com conteúdo confidencial e números de cartão de crédito, apenas para citar os mais facilmente identificáveis.

Entendendo um pouco mais o seu funcionamento, *"o protocolo SSL provê a confidencialidade (privacidade) e a integridade de dados entre duas aplicações que comuniquem pela Internet. Isso ocorre através da autenticação das partes envolvidas e da cifra dos dados transmitidos entre as partes.*

Esse protocolo ajuda a prevenir que intermediários entre as duas pontas da comunicação tenham acesso indevido ou falsifiquem os dados transmitidos."[5]

A autenticação utilizada pelo protocolo SSL baseia-se na utilização de um certificado digital, geralmente armazenado em um arquivo, que contém informações referentes à entidade (que pode ser uma empresa, pessoa, aplicação ou mesmo um computador) para qual ele foi emitido, além da chave pública desta entidade, de modo a identificá-la unicamente dentro de um processo de autenticação.

Esta chave pública é utilizada durante o processo de estabelecimento da conexão, a fim de trocar algumas informações entre o cliente e o servidor, dentre elas a chave secreta que ambos utilizarão durante as trocas de pacotes subsequentes.

É importante ressaltar que o processo de *handshake* utiliza criptografia assimétrica; uma vez estabelecida a conexão, e durante as demais trocas de pacotes, a criptografia simétrica passa a ser utilizada.

O processo de conexão, chamado de **TLS Handshake**, é representado pela figura 1:

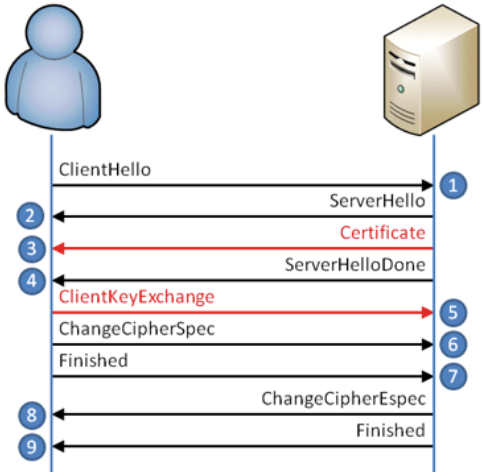


Figura 1 – Handshake de iniciação do protocolo SSL/TLS, baseado em [6]

Pode-se notar, nas linhas 3 e 5, a troca das chaves entre as partes.
Agora que temos uma noção básica de como o protocolo se comporta, vamos falar sobre a falha encontrada.

A vulnerabilidade foi introduzida no código há aproximadamente 2 anos (não vale a pena discutir se a inserção foi proposital ou apenas falha de programação), porém não se sabe ao certo por quanto tempo ela foi utilizada para exploração de ambientes vulneráveis.

A falha ocorre em um determinado trecho do código do OpenSSL para checar se ambos os lados da comunicação estão vivos (é importante ressaltar que **não se trata de um bug ou backdoor na implementação da criptografia no protocolo SSL/TLS**, e sim no controle da comunicação entre as partes). Como o processo de estabelecimento da conexão e/ou da comunicação entre as partes pode sofrer com atrasos, seja por conta da rede ou de algum de seus elementos (roteadores, proxies e outros), os desenvolvedores valeram-se da extensão *heartbeat*, onde as partes envolvidas verificam se a outra parte permanece "viva" de modo que, se não houver resposta de um dos lados, a conexão é encerrada.

Exemplificando, o cliente (que é você) envia um determinado conteúdo dentro do pacote de *heartbeat* para o servidor (que pode ser do seu banco, por exemplo) e o servidor entrega **o mesmo conteúdo** de volta. Se algo der errado durante uma transação (por exemplo, se você fechar o browser porque o site do banco está demorando a responder), o outro lado vai saber (através dos dados do *heartbeat,* que ficarão fora de sintonia) e encerrará a conexão.

A estrutura sugerida pela RFC 6520[8] para determinar o pacote do *heartbeat* é composta por 4 partes: *type* (que pode ser *heartbeat_request* para requisitar ou *heartbeat_response* para responder), o *payload_length* (um inteiro sem sinal de 2 *bytes* usado para indicar o tamanho do *payload* enviado na requisição), o *payload* e *padding*. A extração desta estrutura pode ser vista na listagem 1:

```
struct {
            HeartbeatMessageType type;
uint16 payload_length;
opaque payload[HeartbeatMessage.
payload_length];
opaque padding[padding_length];
} HeartbeatMessage;
```

Listagem 1 – Estrutura do pacote heartbeat

"A vulnerabilidade foi introduzida no código há aproximadamente 2 anos, porém não se sabe ao certo por quanto tempo ela foi utilizada para exploração de ambientes vulneráveis"

A falha no código é conhecida (e considerada clássica) no desenvolvimento: trata-se de um *buffer over-read bug*, cuja explicação mais simples é a de poder ler os dados que estão além do final do *buffer* que foi especificado na memória.

A RFC diz que a resposta do servidor deve conter o *payload* recebido no *request*. Porém, no momento de codificar a extensão, o programador não previu testar se o valor que veio no campo *payload_length* é mesmo o tamanho da mensagem encaminhada no campo *payload*.

Assim, se o lado cliente manda um pacote pequeno de *heartbeat* e insere o valor 0xFFFF no campo *payload_length*, a função vai copiar da memória 65.535 *bytes* e retornar ao cliente, sem reclamar.

Colocando de uma forma mais didática, vamos acompanhar as figuras 2 e 3:

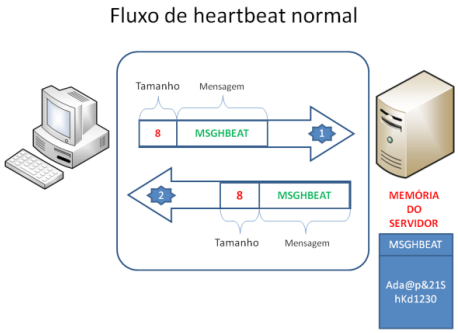


Figura 2 – Fluxo normal de Heartbeat, baseado em [9]

No passo 1 do fluxo normal (figura 2), o cliente envia o request passando como mensagem a palavra "MSGHBEAT" e setando o tamanho para 8. Cabe ressaltar que, normalmente, o *payload* é composto de 2 *bytes* fixos sequenciais seguidos de 16 *bytes* aleatórios. O OpenSSL recebe este *request* e retorna um outro pacote, contendo a palavra enviada (MSGHBEAT) e o tamanho setado para 8 (passo 2).

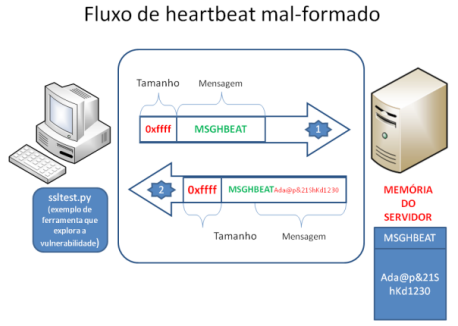


Figura 3 – explorando a vulnerabilidade, baseado em [9]

A vulnerabilidade, conforme já explicado anteriormente e ilustrada pela figura 3, consiste em enviar um pacote com uma mensagem pequena (neste caso, MSGHBEAT no passo 1) mas com o valor máximo que o campo tamanho suporta (em hexa: 0xffff). Ressalta-se que, para realizar o ataque, só é realmente necessário definir os 2 primeiros campos, *type* e *palyload_length*. A versão do OpenSSL que está vulnerável devolve a palavra chave e tudo o que estiver na memória subsequente à ela, até o limite de 65.535 *bytes* (passo 2). Não se tem precisão de quais informações podem ser capturadas, porém a experiência mostra que fazendo-se diversas requisições, onde em cada uma o atacante consegue obter cerca de 64kb de memória do servidor, pode-se recuperar certificados, contas de usuários de sistemas que o servidor hospeda, chaves privadas do certificado e outras informações sensíveis.

Agora, vamos imaginar as consequências disso para usuários de bancos, serviços de e-mail, armazenamento de fotos: com a chave privada na mão, é possível realizar ataques de *Man-in-the-middle* (forma de ataque em que os dados trocados entre duas partes, por exemplo, você e o seu banco, são de alguma forma interceptados, registrados e possivelmente alterados pelo atacante). Há diversas outras informações que podem ser capturadas: tudo que estiver na memória naquele momento como, por exemplo, credenciais, documentos, emails e etc. O céu é o limite!

Vamos extrapolar um pouco mais: e se ficarmos coletando informações de um appliance de VPN que usa a versão do OpenSSL que está vulnerável? Neste caso, podemos obter, por exemplo, credenciais válidas de acesso à rede corporativa, podendo então utilizá-las a fim de se obter uma conexão válida com esta rede. Normalmente, um serviço de VPN é configurado para registrar as atividades em log; ainda assim, uma conexão com esta característica (credenciais válidas) demora mais tempo para ser detectada do que se tentássemos fazê-lo utilizando-se alguma outra técnica (ataque de força bruta, por exemplo).

A correção para este problema é muito simples, porém requer alguns cuidados: deve-se realizar o upgrade da versão do OpenSSL para a mais recente, que corrige esta falha no código, solicitar a troca das credenciais de acesso de todos os usuários que utilizam este ativo, além de revogar **todos** os certificados instalados nestes servidores, solicitando novas emissões. Sim, **é isso mesmo!** Não é apenas comprar novos: temos que revogar os atuais! Não, eu não trabalho em nenhuma empresa que vende certificados. "- E por que cargas d'água você está recomendando isso"? Simples: A falha pode ter sido explorada desde que foi introduzida no código, logo não podemos afirmar que nossos certificados, credenciais de acesso, e-mails e etc não tenham sido coletados e estejam sendo usados para outros fins.

Para finalizar este artigo, vamos abordar a questão crucial: "- E como eu convenço os executivos

da minha empresa acerca dos problemas causados por essa vulnerabilidade para que suportem o plano de ação que estabeleci (inclusive com verba para comprar novos certificados, se for o caso – algumas autoridades certificadoras se propuseram a fazer a revogação e nova emissão sem custo) para eliminá-la do meu ambiente?".

O primeiro passo, e o mais importante na minha opinião, é não fazer FUD (Fear, Uncertanly and Doubt – termo usado para definir abordagens que pressionam os executivos pela ameaça e por gerar incerteza) – a chance de se ganhar um sonoro **NÃO** é grande. A abordagem que costuma dar certo comigo consiste em você embasar seu discurso naquilo que o executivo entende, de forma direta e pragmática: explicar que existe um problema (neste caso, que não afeta apenas a nossa empresa), que já temos a solução e um plano para a execução, que o time já está mobilizado para realizar as atividades, que haverá um custo (se realmente houver) que precisa ser aprovado e qual o impacto (financeiro – mais palpável para eles - ou indireto) que eliminaremos com a ação em curso.

Importante: não peça permissão, seja proativo e diga que já está executando as correções necessárias. Nestas horas, ser diretivo dá mais confiança ao executivo de que você está no caminho certo e de que ele não precisará entender

"O primeiro passo, e o mais importante na minha opinião, é não fazer FUD - a chance de se ganhar um sonoro NÃO é grande"

o problema e discutir a solução com você – você foi contratado para solucionar problemas, não ficar compartilhando com ele as possíveis soluções!

Para não ficar nestas fórmulas mágicas que todos nós vemos por aí que parecem mais auto-ajuda (não menosprezo nenhum tipo de literatura a este respeito, só acho que, na maioria das vezes, o cenário desenhado por este tipo de literatura não condiz em nada com nossa realidade), vou colocar na prática a estratégia que usei para ganhar o suporte e a confiança dos executivos da empresa que trabalho para realizar a atualização dos servidores e assegurar a compra de novos certificados.

Tão logo eu recebi a informação desta vulnerabilidade, pesquisei na Internet (fóruns, sites de notícia, site da comunidade mantenedora do OpenSSL, o próprio CVE do MITRE, blogs, etc) tudo que pudesse me clarificar do que se tratava e quais as medidas corretivas necessárias para eliminá-la ou, ao menos, controlá-la.

Li diversas discussões a respeito do que poderia ser capturado com este ataque (Conseguimos capturar a chave privada ou não? Dá acesso ao arquivo de logins ou senhas ou não? Que outros dados são passíveis de vazamento?), quais ambientes estão comprometidos (Linux, Unix, Windows, Mac OS). Em seguida, e com base nas informações de ambientes comprometidos e o que poderia ter sido capturado (assumi que o certificado era uma das possíveis informações vazadas), estabeleci uma lista dos ativos que poderiam estar comprometidos e qual(is) processo(s) crítico(s) dependia(m) destes ativos.

Com a lista estabelecida, entrei em contato com o time que suporta a operação do Datacenter e solicitei que: (1) confirmassem qual a versão do OpenSSL que estava sendo executada (minha diretiva neste ponto foi clara: deveríamos saber todos os servidores que estavam com o OpenSSL vulnerável – (IMPORTANTE: versões anteriores a versão 1.0.1 não estão vulneráveis à falha do *HeartBleed* especificamente, mas podem estar vulneráveis a inúmeros outros problemas!) , não apenas os que foram identificados anteriormente) e (2) executassem uma mudança emergencial para realizar o upgrade da versão do OpenSSL, começando obviamente pelos ativos que suportavam processos críticos mas não se limitando a eles, uma vez que manter o ambiente com

uma versão desatualizada em servidores não críticos pode se tornar um ponto de partida para se obter informações para atacar o restante do ambiente.

Em paralelo, solicitei a nossa parceira emissora de certificados que iniciasse o processo de geração de novos certificados (como não tínhamos muitos, o custo foi, de certa forma, irrisório), que envolviam a aprovação dos executivos (a famosa validação telefônica).

Uma vez acertado com a emissora, fui até o Comitê Executivo e expliquei, sem entrar em detalhes técnicos, que havíamos descoberto uma falha em um componente de mercado que era responsável por garantir o sigilo de nossas transações, que este componente era largamente usado (inclusive por nossos concorrentes) e que, por precaução, devíamos trocar nossos certificados (nas minhas palavras, "aquele arquivo que permite fechar o cadeado no Internet Explorer quando você está acessando o site do banco e que garante que ninguém além de você está vendo suas transações") atuais por novos. Por este motivo, eu precisaria da disponibilidade da agenda deles para fazer a validação por telefone para emitir os novos certificados e que o custo seria absorvido por nós, economizando algum dinheiro em alguma negociação com fornecedores para renovar licenças. Deixei claro a eles que esta medida era preventiva, porque a correção em si já estava sendo conduzida pelo time do Datacenter e que, tão logo fosse finalizada, eles seriam comunicados a respeito.

Com a aprovação deles, restou apenas coordenar as atividades de substituição e revogação dos certificados nos ambientes.

Resumo: em menos de uma semana estávamos com o parque atualizado, certificados trocados e os executivos cientes que eu e o time da TI fizemos o que tinha que ser feito.
Vejam que, na abordagem usada, em nenhum momento eu usei termos técnicos ou entrei no detalhe de como a falha ocorria e era explorada; apenas expus, com fatos e argumentos não-técnicos, o que precisava ser feito.

O resto é história.

[1] - *http://www.openssl.org/*, acessado em 10/06/2014
[2] - *http://www.openssl.org/news/secadv_20140407.txt*
[3] - *http://heartbleed.com/*, acessado em 10/06/2014
[4] – *www.openssl.org/about/*, tradução livre em 15/06/2014.
[5] - *http://pt.wikipedia.org/wiki/Transport_Layer_Security*, acessado em 15/06/2014
[6] *https://devcentral.f5.com/articles/ssl-profiles-part-1*
[7] –*CVE* – acrônimo de Common Vulnerabilities and Exposures, é um dicionário de nomes comuns (identificadores) para vulnerabilidades de segurança da informação para fins de conhecimento público. A coordenação do CVE pertence ao MITRE, uma organização sem fins lucrativos que opera centros de pesquisa e desenvolvimento patrocinados pelo governo federal dos EUA.
[8] *http://tools.ietf.org/html/rfc6520*
[9] *http://fronterahouse.com/blog/ultimate-heartbleed-guide-for-non-techies/*

JULIO CEZAR DE RESENDE MOREIRA

Tem 39 anos, formado em Tecnologia de Processamento de Dados pela Univ. Mackenzie em 1996 e pós-graduado em Gestão Empresarial pela FEI em 2006, trabalha com Tecnologia da Informação desde os 16 anos. Começou sua carreira atuando em desenvolvimento de sistemas e teve passagens por suporte técnico e arquitetura. Atua com segurança da informação desde 2001. Atualmente é gerente de segurança numa multinacional francesa do varejo. É apaixonado por tecnologia, artes marciais e música (esta última como ouvinte apenas).
Possui certificações CISSP, ISO 27001 Lead Auditor e MCSO.


11ª EDIÇÃO 2014
H2HC
HACKERS TO HACKERS CONFERENCE
FALTA POUCO PARA A PRÓXIMA H2HC !
VOCÊ ESTÁ PRONTO PARA A SUA MAIOR AVENTURA TECNOLÓGICA NA SELVA?
GARANTA JÁ SEU INGRESSO EM NOSSO SITE
www.h2hc.com.br

Segurança de *Software Open Source* através de Análise Estática.

POR RAGHUDEEP KANNAVARA

Para maiores detalhes, bem como referências à afirmações feitas neste artigo, o leitor pode consultar a fonte que deu origem ao texto: [1].

Apesar da SCA (*Static Code Analysis* - Análise Estática de Código) não ser uma nova tecnologia, ela tem recebido muita atenção e está sendo rapidamente adotada como uma atividade fundamental no Ciclo de Desenvolvimento de *Software*, visando melhorar a qualidade, confiabilidade e segurança do software. A maioria dos estabelecimentos de desenvolvimento de *software* têm uma equipe dedicada de profissionais de validação de *software*, onde entre as responsabilidades está incluída a execução de ferramentas de análise estática no *software* em desenvolvimento, seja em um ambiente ágil ou em um modelo em cascata tradicional. Ferramentas de SCA são capazes de analisar o código desenvolvido em diversas linguagens de programação e *frameworks* que incluem e não se limitam a Java, .Net e C/C++.

Seja o objetivo desenvolver um aplicativo de *software* ou um *firmware* embarcado de missão crítica, ferramentas de análise estática provaram ser um dos primeiros passos para a identificação e eliminação de erros de *software*, sendo fundamentais para o sucesso global do projeto de *software*. Licenciamento e aplicação exaustiva de ferramentas de análise estática em um projeto de desenvolvimento de *software* são tarefas caras, mas são geralmente utilizadas por empresas de *software* proprietário para melhorar a qualidade do produto.

O que então permanece como um vazio expansivo e inexplorado são os domínios intermináveis de códigos open source, geralmente vistos como tendo sido bem revisados por terem sido utilizados e reutilizados em um número incontável de aplicações por inúmeras instituições acadêmicas e comerciais em todo o mundo. 'Expansivo' indicando que está constantemente adicionando novos códigos fonte e 'Inexplorado' indicando a ausência de uma entidade imparcial dedicada a analisar estaticamente cada linha de código *open source*.

Adicionalmente, nos últimos tempos, os projetos de código aberto têm ciclos de lançamento relativamente mais rápidos em comparação com código fechado. Por exemplo, o Android tem lançamentos a cada 3 meses enquanto o Windows a cada 3 anos. A taxa de inovação que está sendo conduzida pelo *open source* através das indústrias *mobile*, *cloud* e *big data* é enorme.

Mesmo empresas de SCA como Klocwork e Coverity terem tomado iniciativa em analisar códigos open source e compartilhar os resultados com a comunidade, a taxa com que novas versões de *software open source* são lançados ou atualizados apresenta uma barreira a estes esforços. Embora tal esforço seja, por muitos, atribuído à paranóia, é evidente que a crise financeira e a diminuição de verba de TI levaram o *software open source* a entrar em aplicações comerciais, por exemplo, Linux, Apache, MySQL e OpenSSL. Além disso, esforços como os programas do Centro de Software Assegurado (CAS) da NSA que apresentaram um estudo de ferramentas de análise estática para C/C++ e Java no ano 2010 usando casos de teste disponíveis como Juliet Teste Suites, são um passo na direção correta, mas, mesmo assim, não há métricas absolutas para a escolha de uma ferramenta SCA particular.

Na maioria das instituições de desenvolvimento e teste de *software*, apesar do produto de *software* como um todo ser submetido a análise dinâmica, como *fuzzing* ou teste de caixa preta, e o código de *software* recentemente desenvolvido ser submetido a uma rigorosa análise estática, códigos *open source* incorporados no *software* geralmente não estão sujeitos à mesma rigorosa análise estática. Isto pode estar baseado no pressuposto de que o *software* de código aberto é mais seguro e possui menos *bugs* do que o *software* de código fechado devido ao fato de o código fonte estar disponível de forma

gratuita e de haver muitas pessoas olhando para ele. Outra razão é que pode haver restrições de licença relacionadas à alterações realizadas em código *open source*.

Apesar do fato de essa suposição ter resistido ao teste do tempo, e é provável que não possa ser provada como errada, na realidade, e não surpreendentemente, executando uma ferramenta popular de SCA contra uma base de código open source muito popular como o *kernel* do Linux, v2.6.32.9, revela-se que um determinado subconjunto das vulnerabilidades publicadas no NVD (National Vulnerability Database), após o lançamento desta versão do *kernel*, poderia ter sido detectado muito mais cedo caso a análise estática diligente tivesse sido executada anteriormente, por exemplo, antes do lançamento do *kernel*. O número de vulnerabilidades encontradas no kernel do Linux por análise estática, juntamente com suas áreas, estão ilustradas na Figura 1.

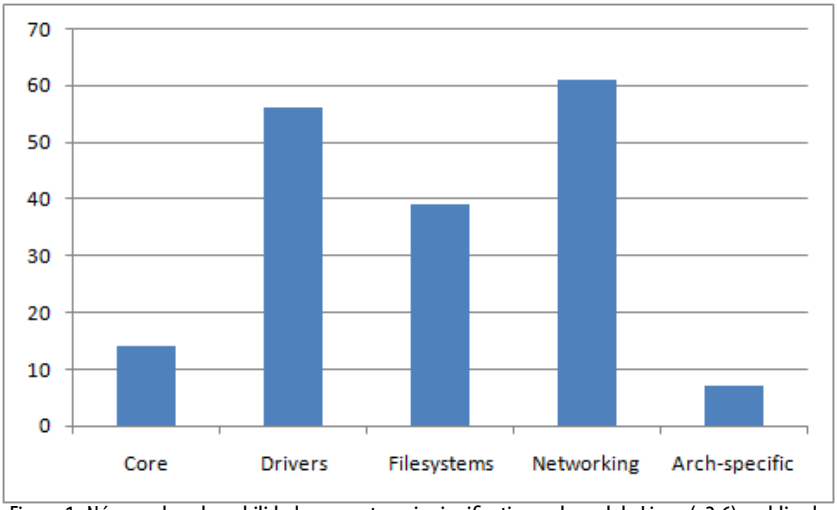


Figura 1- Número de vulnerabilidades por categoria significativa no kernel do Linux (v2.6), publicada no NVD (2010-11).

Tendo dito isto, ressalta-se que as vulnerabilidades detectadas somente por SCA não serão abrangentes, uma vez que algumas falhas são dependentes de *run-time*, mas uma certa classe de vulnerabilidades pode ser detectada por SCA, incluindo *buffer overflows*, *integer overflows/underflows*, erros de sinais de inteiros e inicialização imprópria de memória. Estas são algumas das vulnerabilidades que têm sido frequentemente exploradas para lançar ataques maliciosos em diversas aplicações de *software*.

Assim, observado o fluxo de trabalho convencional, ilustrado na Figura 2, que sujeita o código recém-desenvolvido para uma rigorosa SCA enquanto o código *open source* geralmente não, e, assim, ter afirmado que uma certa classe de vulnerabilidades pode ser detectada precocemente por SCA, o que naturalmente inclui vulnerabilidades em código aberto, torna-se óbvio que é benéfico incluir a saída da ferramenta SCA tanto para *software* recentemente desenvolvido quanto para *software* open source adotado, no pacote de

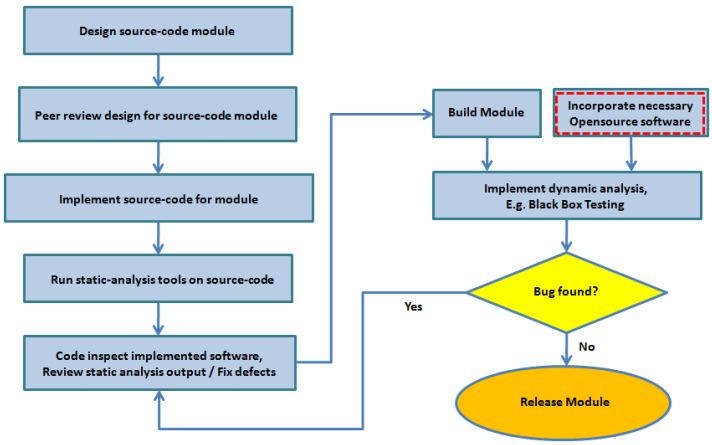


Figura 2 - Processo de desenvolvimento de software convencional

revisão formal de *software*. Qualquer *bug* de *software* encontrado pela análise estática é corrigido em ambos os códigos, *open source* e recém desenvolvido, antes de ser submetido a análise dinâmica e bem antes de ser eventualmente lançado para o mercado. Este processo pode não encontrar todos os *bugs*, mas vai ajudar a pegar alguns erros no início, proporcionando uma oportunidade para corrigi-los mais cedo. Mas, em praticidade, embora a análise estática de código open source adotado seja útil na identificação de certos *bugs* de *software* desde o início, há desafios e *trade-offs* técnicos e de projeto que podem não fazer desta uma proposição viável em determinadas

situações, como discutido mais adiante. Ferramentas SCA tendem a produzir um grande número de falso-positivos. Rever e eliminar falso-positivos podem ser tarefas intimidadoras tanto para as equipes *quality assurance* quanto para as de desenvolvimento, especialmente quando os projetos estão enfrentando severas restrições de recurso, tempo e orçamento. Além disso, falso-positivos podem ser reais ou perceptuais. A primeira categoria são casos em que a ferramenta apenas está errada, precisa ser ajustada ou o verificador necessita somente ser desligado para aquela particular base de código.

A outra categoria de falso-positivos (perceptual) é mais um problema de educação e/ou priorização do desenvolvedor. Muitas vezes, especialmente no caso de falhas de segurança ou erros de confiabilidade mais sutis, a ferramenta está correta, mas o desenvolvedor não compreende por que é um problema ou não se preocupa tanto com o problema específico, ou seja, educação ou priorização. Além disso, diferentes vendedores de SCA tendem a encontrar problemas diferentes em uma mesma base de código, mas essa pesquisa detectou que utilizando 3 diferentes ferramentas de análise estática pode-se garantir uma boa cobertura de código. Compete ao fornecedor das ferramentas garantir que os relatórios de *bugs* gerados sejam claros, mas a outra parte da equação é o sensível processo de utilizar desenvolvedores mais experientes dentro da organização para auxiliar na análise, bem como providenciar treinamento adequado a quem for necessário. No entanto, uma vez que os falsos positivos são eliminados, comunicar os problemas genuínos encontrados para as equipes de desenvolvimento corrigirem faz-se necessário. Às vezes, isso pode exigir as correções serem comunicadas aos mantenedores do *software open source* antes do lançamento do produto, com base nos termos da licença do *software open source*.

Uma vez que o processo com segurança reforçada para incorporação de *software open source*, ilustrado na Figura 3, envolve uma enorme quantidade de esforço e é relativamente desafiador, uma forma de endereçar este desafio poderia ser identificar os componentes críticos (componentes com histórico de vulnerabilidades) do projeto e concentrar esforços de SCA sobre eles, por exemplo, componentes de 'rede'.

Outra forma poderia ser se concentrar em componentes de maior complexidade no código *open source* adotado. Arquivos de maior complexidade têm maior probabilidade de gerar erros durante a correção, melhoria ou refatoração do código fonte. Na maioria dos projetos de código aberto, a maior parte do esforço de desenvolvimento é comunicado através de listas de discussão. Os desenvolvedores estão espalhados por todo o mundo e têm diferentes níveis de habilidades de desenvolvimento de *software*. É uma tarefa desafiadora para qualquer entidade central responsável pela coordenação dos esforços

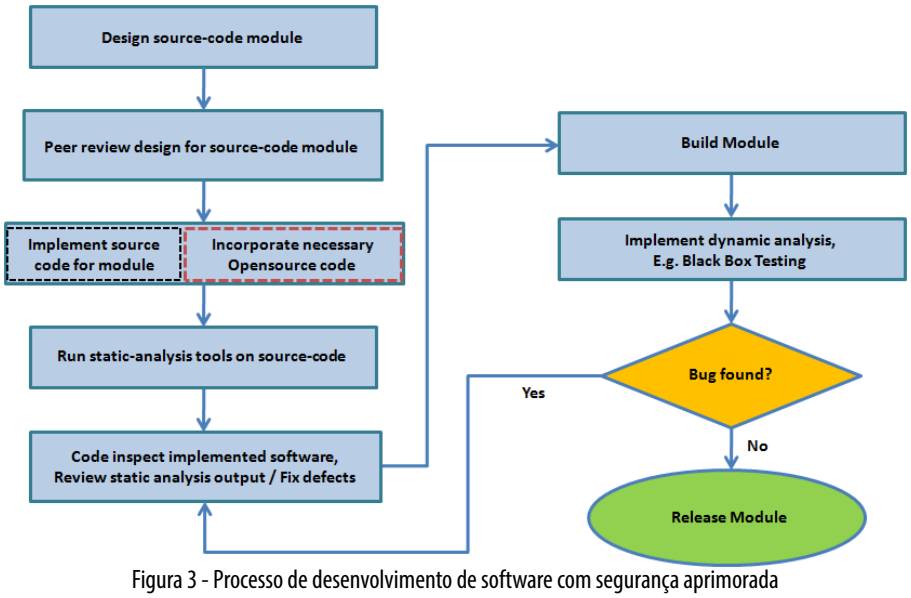


Figura 3 - Processo de desenvolvimento de software com segurança aprimorada

de desenvolvimento. Concentrar SCA nestes componentes de maior complexidade ou criticidade pode economizar verba, tempo e recursos por reduzir o escopo da revisão. Além disso, somente SCA não é suficiente; pode-se solicitar revisão de código, *fuzzing* e outras atividades de validação para melhorar a segurança de *software*.

Em resumo, fortificar o código open source que é incorporado em um *software* comercial é tão importante quanto fortificar o *software* proprietário em si. As linhas entre projetos comerciais e projetos *open source* estão ficando mais fracas devido ao aumento contínuo da adoção do *software* de código aberto no desenvolvimento de *software* comercial. O Gartner prevê que, até 2014, 50% das organizações Global 2000 vão vivenciar desafios de tecnologia, custo e segurança pela falta de governança open source. *Open source* é onipresente, é inevitável; ter uma política contra o *open source* é impraticável e coloca a organização em desvantagem competitiva.

Por outro lado, a necessidade imediata é ter políticas para obter formalmente *software open source* e entender, gerenciar e mitigar seus riscos de adoção. Juntamente com a validação cuidadosa de *software* proprietário recentemente desenvolvido utilizando SCA, também é altamente desejável incluir análise estática de qualquer código *open source* necessário para o produto no processo de validação geral do *software*. Uma analogia pode ser feita com a abordagem de *build*. Ao utilizar uma biblioteca *open source*, o desenvolvedor deveria baixar um binário e o inclui-lo diretamente no *build* ou baixar o código fonte e compilá-lo nos *milestones* do projeto? Por não se utilizar o binário diretamente e baixar o código fonte e compilá-lo nos *milestones* do projeto, o projeto estaria protegido contra detalhes do ambiente de compilação de terceiros, que podem não ser relevantes ao projeto.

Se assim for, por que não incluir esse mesmo código *open source* na execução da análise estática juntamente com o código recém-desenvolvido? É bem compreendido na indústria de *software* que quanto mais cedo um *bug* possa ser detectado, mais barato será para corrigi-lo, evitando assim caras ações judiciais ou danos irreparáveis à reputação da empresa. De muitas formas, este esforço extra é fundamental para melhorar a qualidade, confiabilidade e segurança gerais do produto de *software*. Resultados recentes indicam que projetos de código aberto com uma comunidade de desenvolvimento ativa e com comprometimento com a qualidade estão implementando com sucesso análise estática e outras medidas de controle de qualidade da mesma forma que projetos comerciais. Por fim, as organizações que podem arcar com o custo de ferramentas SCA e das mudanças no processo, e têm uma forte necessidade de segurança, encontrarão valor neste esforço.

Referências:
[1] Raghudeep Kannavara, "Securing Opensource Code via Static Analysis", 2012 IEEE Sixth International Conference on Software Testing, Verification and Validation, pp. 429-436.

RAGHUDEEP KANNAVARA

Raghudeep Kannavara atualmente trabalha como Security Architect/ Researcher e Technology Analyst na Intel Corp., Hillsboro, OR, USA. É especialista em threat assessment, vulnerability assessment, validação de segurança, security development lifecycle e design for security. Possui PhD em Engenharia da Computação e tem publicado extensivamente no IEEE sobre diversos temas de segurança.

CURIOSIDADES

FOREWORDS

Por Gabriel Negreira Barbosa

A partir de agora, as edições da H2HC Magazine contam com a coluna "Curiosidades". A principal ideia dessa coluna é abordar temas curiosos de maneira descomplicada. Desta forma, serão discutidos somente os detalhes estritamente necessários para a clara compreensão do leitor.

Convidamos os leitores a enviar sugestões de assuntos a serem abordados nas edições futuras. Para os interessados em se aprofundar nos temas tratados, não deixem de nos escrever: será um prazer trocar ideais e compartilhar material mais detalhado a respeito! **Esperamos que gostem! Boa leitura!**

O *HOST* ESTÁ REALMENTE INDISPONÍVEL?

POR YGOR DA ROCHA PARREIRA

Em um ambiente de rede local é comum testar a disponibilidade de um determinado equipamento utilizando o comando *ping*. Quando não há resposta assume-se que o equipamento está desligado ou indisponível. Mas será que isto é sempre verdade? Podemos realmente assumir que estando em um mesmo ambiente de rede local[1], quando um determinado *host* não responde ao comando *ping* é porque ele encontra-se indisponível? O que mais poderíamos verificar para chegar a uma conclusão mais acertada?

Os sistemas operacionais mais comumente encontrados em ambiente de rede local são alguns dos sabores de Windows. Estes sistemas operacionais já vem com um *firewall* ativo, por padrão já com algum comportamento pré-definido. Nas versões mais recentes do *Windows*, por padrão este comportamento é definido de acordo com o perfil de classificação da rede (pública, particular, etc), onde quando a rede é definida como sendo pública o sistema operacional bloqueia as solicitações feitas pelo comando ping. Para exemplificar, a *Figura 1* mostra um ambiente típico de rede local, onde temos um *host Windows 8.1* que vai ser testado a partir de outro *host Windows XP*.

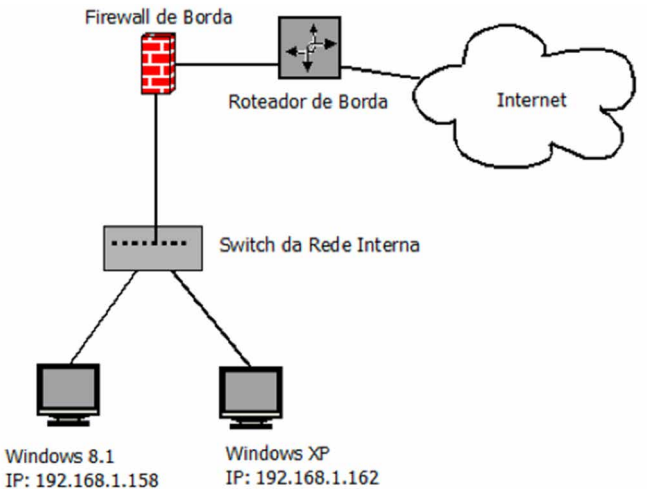


Figura 1 – Topologia do ambiente testado

A Listagem 1 mostra o *host* 192.168.1.162 executando o comando *ping* contra o *host* 192.168.1.158 *(Windows 8.1*), onde não é obtida resposta do *host*.

```
C:\Documents and Settings\dmr>ping 192.168.1.158

Disparando contra 192.168.1.158 com 32 bytes de dados:

Esgotado o tempo limite do pedido.
Esgotado o tempo limite do pedido.
Esgotado o tempo limite do pedido.
Esgotado o tempo limite do pedido.

Estatísticas do Ping para 192.168.1.158:
    Pacotes: Enviados = 4, Recebidos = 0, Perdidos = 4 (100% de perda),

C:\Documents and Settings\dmr>
```

Listagem 1 – Testando com o comando ping contra o host Windows 8.1

[1] *-Por ambiente de rede local etentende-se um conjunto de máquinas no mesmo segmento físico e lógico, sem roteadores e firewalls de rede no meio do caminho*

Sabendo que os dois *hosts* estão ligados e ativos na rede, por que o *Windows 8.1* não responde? Isto acontece porque o *firewall* do sistema operacional bloqueia as solicitações de *ping*. Então o que mais podemos verificar para checar a disponibilidade do *host Windows 8.1*? Conhecendo como ocorre uma comunicação via rede, sabemos que antes dos pacotes do *ping* serem enviados, pacotes de resolução de endereços físicos (endereços *MAC – Media Access Control*) devem ser trocados. Neste caso, o endereço *MAC* resolvido é o do *host* testado, pois se encontram no mesmo segmento de rede físico e lógico. Apesar da maioria dos *firewalls* (inclusive o do *host* testado) terem a capacidade de filtrar estes tipos de pacotes, quase nunca o fazem.

Assim sendo, uma entrada é inserida na tabela de cache ARP local do *host* que executou o comando *ping*, esta tabela pode ser consultada para determinar se o *host* testado realmente se encontra indisponível. A Listagem 2 mostra esta verificação:

```
C:\Documents and Settings\dmr>arp -a

Interface: 192.168.1.162 --- 0x2
  Endereço IP         Endereço físico     Tipo
  192.168.1.158       a4-1f-72-f5-cd-17   dinâmico

C:\Documents and Settings\dmr>
```

Listagem 2 – Verificando a tabela *cache* ARP local do Windows XP

É possível notar que apesar do *Windows 8.1* não responder ao comando *ping*, o *host* se encontra ativo visto que respondeu a solicitação de resolução de endereço físico.

Caso o *host* testado (*Windows 8.1*) estivesse de fato indisponível, o mapeamento físico não apareceria na tabela de *cache* ARP. Quando o sistema que realiza o teste é um *Linux*, para este mesmo comportamento (*host* indisponível), é criada uma entrada do tipo <*incomplete*> na tabela de *cache* ARP local.

YGOR DA ROCHA PARREIRA

Ygor da Rocha Parreira faz pesquisa com computação ofensiva, trabalha como consultor de segurança de aplicações na Trustwave e é um cara que prefere colocar os bytes à frente dos títulos.

ENGENHARIA REVERSA DE SOFTWARE

FOREWORDS
Por Fernando Mercês

Imediatamente após terminar a leitura da nova coluna "Fundamentos para Computação Ofensiva", pensei na existência de uma coluna com uma didática similar, mas com foco em engenharia reversa. Certamente as duas colunas juntas se reforçariam, já que o aprendizado de ambos os assuntos podem ter sequência juntos. Pois aqui estamos, com a nova coluna sobre engenharia reversa de *software*. Tenho a honra de inaugurá-la e o compromisso de conduzir o leitor nesse estudo dirigido.

Por que estudar engenharia reversa?

"Algumas pessoas vêem coisas que existem e perguntam, Por que?
Algumas pessoas sonham com coisas que nunca existiram e perguntam, Por que não?
Algumas pessoas precisam ir pro trabalho e não têm tempo para isso tudo."
(George Carlin)

Mas se queres motivos, posso citar vários. O aprendizado de como um programa *realmente* se comporta ao ser executado no sistema operacional é um deles. Outra utilidade é reimplementação de recursos, por exemplo, suponha que um programa proprietário salve um formato de arquivo também proprietário. Formato este que você gostaria de abrir (ler o arquivo) sem ter de pagar por um programa só para isso, já que seu interesse está no conteúdo do arquivo, não no *software* que o mostrará. Com engenharia reversa você pode verificar o algoritmo que o programa proprietário segue para abrir o tal formato e implementar seu próprio algoritmo similar, de forma livre. Agradeça à engenharia reversa por ter clientes de IM (*Instant Messenger*) livres que se comunicam com várias redes de mensageria, por abrir arquivos do Office fora dele e por mais centenas de recursos implementados em diversos *software* e *hardware*, livres ou não.

Em segurança, um uso comum é a análise de vulnerabilidade (sem o código) e a análise de *malware*. O mercado está em alta. O *cracking* (neste caso, remoção de proteção contra cópias ilegais) de *software* também tem sua base na engenharia reversa, no entanto vale lembrar que em alguns países é criminalizado como pirataria.

A engenharia reversa é nomeada de tal forma porque, com ressalvas, é como *"ler, e poder alterar o código fonte do programa sem tê-lo"*. Entre aspas porque na realidade o código fonte não vai para o programa compilado.

Os exemplos citados aqui não esgotam suas possibilidades de uso. Como sempre *"Criatividade é mais importante que conhecimento"* (Einstein), ou seja, o que você pode fazer com engenharia reversa (e com outras artes) só depende de você. Não há limites. E sim, a ER (como a chamarei algumas vezes daqui em diante) é uma arte. É possível criar a partir do criado, reverter, distorcer, adicionar, remover, refazer o que foi feito. O engenheiro reverso é, com efeito, um artista. :)

INTRODUÇÃO À ENGENHARIA REVERSA

POR FERNANDO MERCÊS

"Não sabendo que era impossível, foi lá e fez"
(Jean Cocteau/Mark Twain)

Tratando-se de linguagens compiladas (C, C++, Delphi, etc), o processo prático de criação de um programa é, basicamente:

1. Escrita de código (em texto mesmo).
2. Compilação.
3. Link-edição (*linking*).

Quando o foco é a criação de um programa para ser executado nos principais sistemas operacionais para PC, o resultado final deste processo é, geralmente, um executável ou uma biblioteca. E assim um *software* é distribuído, podendo ter vários executáveis, bibliotecas e outros arquivos em seu pacote

de distribuição, e/ou ainda um programa de instalação. Sem acesso a seu código fonte (o que normalmente acontece quando o software é proprietário), como saber exatamente o que o programa faz? Como garantir que o binário faz realmente o que o código-fonte diz?

A maioria dos desenvolvedores de software concentra seus estudos na primeira etapa do processo descrito (escrita de código) já que as outras duas são de responsabilidade do ambiente de compilação. De fato, a maioria dos programadores não faz ideia do que é um executável e de que ele pode ser revertido (analisado por engenharia reversa) e até alterado.

Mas é possível saber o que o programa faz e como faz. Tudo com engenharia reversa. Vamos analisar um primeiro programa que faz, simplesmente, nada. :)

Usando uma distribuição *Linux* qualquer com o gcc, crie um arquivo nada.c com o seguinte conteúdo (testado com Debian 7 64-bits, kernel 3.8.0, GNU Make 3.81 e gcc 4.7.3).

```
int main() { return 9; }
```

E depois execute:

```
$ make nada
cc    nada.c  -o nada
```

Como pode ver, o ambiente de compilação é chamado para compilar o arquivo nada.c e gerar o executável nada. No *Linux*, normalmente o cc é um link para o compilador gcc. Nesta construção, o make não precisa de um *Makefile*. ;)

O que você acabou de fazer em uma linha envolve processos bem complexos. Após validar toda a sintaxe da linguagem C, o ambiente de compilação gerou um código compreensível para o processador da arquitetura que você utilizou, resolveu nomes de função (que você tenha escrito, só a main, neste caso) e "linkou" seu programa com a biblioteca padrão da linguagem C (libc). No caso das distros *Linux* convencionais, a libc geralmente utilizada é a *glibc* (GNU C Library).

Além disso, o executável gerado segue um padrão, um formato. No caso dos executáveis no *Linux*, esse formato é geralmente o ELF [1].

Percebe quanto trabalho é feito depois da escrita do código, normalmente tida como término da construção do programa? ;)

O exemplo nada, apesar de meio vago (hehe), é um ótimo exemplo, pois suponha que você se depare com este programa compilado e o execute. Como ele "não vai fazer nada", ficará aquela sensação de "ih.... O que aconteceu?", é ou não é? Nem pense duas vezes, ER nele!

Antes de propriamente reverter o binário, vamos dividir o processo em dois estágios. O *disassembly*, que é o processo de gerar código ASM (como chamarei a linguagem *Assembly* em alguns casos daqui em diante) a partir do código de máquina presente no executável ou biblioteca, e a depuração (ou *debugging*), que é o processo de executar um certo número de instruções, uma de cada vez, a fim de observar o comportamento gradual do programa.

Tenha em mente que o processo de compilação gerou um arquivo binário, ou seja, uma série de *bytes* "um ao lado do outro" no disco. A maior parte desses *bytes* não representa aquela única linha de código fonte do programa "nada", mas sim a estrutura do ELF. Já que este arquivo não passa de um monte de *bytes* (estrutura ELF, código de máquina e etc), vamos olhar tais *bytes* com um visualizador hexadecimal:

```
$ hd nada
00000000  7f 45 4c 46 02 01 01 00  00 00 00 00 00 00 00 00  |.ELF............|
00000010  02 00 3e 00 01 00 00 00  00 04 40 00 00 00 00 00  |..>.......@.....|
00000020  40 00 00 00 00 00 00 00  70 11 00 00 00 00 00 00  |@.......p.......|
00000030  00 00 00 00 40 00 38 00  09 00 40 00 1e 00 1b 00  |....@.8...@.....|
00000040  06 00 00 00 05 00 00 00  40 00 00 00 00 00 00 00  |........@.......|
00000050  40 00 40 00 00 00 00 00  40 00 40 00 00 00 00 00  |@.@.....@.@.....|
00000060  f8 01 00 00 00 00 00 00  f8 01 00 00 00 00 00 00  |................|
<suprimido>
```

O hd (*hexdump*), criado originalmente para o BSD, é um visualizador hexadecimal (não é um editor porque ele somente lê). Dos conceitos de computação, sabemos que um *byte* (na arquitetura Intel IA-32, a qual estamos admitindo aqui) corresponde a 8 *bits*, logo, pode variar de 0 a 255 (assumindo que estamos falando de números inteiros sem sinal), ou seja, 2^8 possibilidades. O que o *hexdump* faz é mostrar, por padrão, 16 *bytes* por linha, cada um separado por espaço e dois grupos de 8 *bytes* separados por 2 espaços, mas ao invés de mostrar em decimal (0-255), ele o faz em hexadecimal, que vai de 00 até FF (em decimal, de 0 a 255, respectivamente). Por que em hexadecimal? Porque dessa forma você precisa de menos caracteres para representar maiores quantidades, já que a base é 16. O próprio 255, por exemplo, precisa de 3 caracteres para ser representado em decimal, enquanto em hexa bastam dois: ff.

Na primeira coluna o hd mostra o *offset* (posição relativa) do *byte*. No exemplo acima, o *byte* na posição relativa 0x60 ao início do arquivo é 0xf8, enquanto o *byte* na posição relativa 0x61 é 0x01. E o *byte* na posição relativa 0x68 é? :)

A última coluna é a representação ASCII [2] dos *bytes* apresentados, quando há. Se um *byte* está na faixa coberta pela tabela ASCII, sua representação ASCII é mostrada, do contrário um ponto é exibido para informar que não há representação ASCII para o *byte* em questão. Veja que o primeiro *byte*, 0x7f, não possui representação ASCII, enquanto os três *bytes* seguintes, 0x45, 0x4c e 0x46, possuem. Consulte uma tabela ASCII online e veja o motivo. ;)

DICA: Para entender mais sobre o funcionamento de um dumper hexa (como também se chamam os visualizadores hexadecimais), estude o código do *hdump* [3], que é livre e muito simples. Ele apresenta uma saída idêntica à do hd, com a vantagem de ser multiplaforma e já possui versões compiladas para *Windows* de 32 e 64-bits. O código é bastante enxuto (cerca de 100 linhas) e está comentado em Português.

A minha versão do binário "nada" ficou com 8.468 *bytes* de tamanho, por isso suprimi a saída do hd aqui no texto. O início do arquivo não representa o início do código do programa em si, mas sim campos inerentes à especificação do formato ELF. Mesmo que visualizemos com o hd os *bytes* referentes ao código de máquina gerado pelo processo de compilação da nossa única linha de código fonte, estes provavelmente não seriam inteligíveis ao serem lidos pois seriam somente uma sequência de números hexadecimais (a não ser que o leitor esteja mapeando cada *byte* com as instruções IA equivalentes). Por isso o processo de *disassembly* é necessário. Para tal, vamos usar agora o *disassembler objdump*:

```
$ objdump -M intel -d nada
```

A opção -M intel do *objdump* configura o disassembler para utilizar a sintaxe Intel – assim como linguagens de programação, o código *assembly* pode ser representado de mais de uma forma. Já a opção -d pede ao *objdump* que faça o *disassembly* somente das seções que contenham código, o que é possível porque o *objdump* conhece a estrutura dos binários ELF. Você vai perceber que há várias funções não colocadas pelo programador (no código fonte), em nosso caso elas fazem parte da libc e de seu processo de inicialização (mais informações podem ser obtidas nos artigos da seção "Fundamentos para Computação Ofensiva"). Mas vamos à main:

```
00000000004004ec <main>:
  4004ec:       55                  push   rbp
  4004ed:       48 89 e5            mov    rbp,rsp
  4004f0:       b8 09 00 00 00              mov    eax,0x9
  4004f5:       5d                  pop    rbp
  4004f6:       c3                  ret
```

Um *disassembler* tem uma saída muito mais interessante para código que um *dumper* hexa. Enquanto este último mostra *offset* dos *bytes*, os *bytes* em si e a representação ASCII deles (se houver), um *disassembler* mostra endereço (endereço base estimado para o binário + *offset*), *bytes* e as instruções que estes *bytes* representam. Tomemos a primeira instrução da função main deste exemplo, "PUSH RBP". Essa instrução é codificada pelo *byte* 0x55 – ele aparecerá muito frequentemente em início de funções.

Já a instrução "MOV RBP, RSP" é representada pela sequência de *bytes* 0x45 0x89 0xe5. E assim sucessivamente.

Estudar engenharia reversa é analisar, dentre outras coisas, códigos *assembly*. Vamos então entender linha a linha:

```
push   rbp          // coloca o valor do registrador RBP na pilha
mov    rbp,rsp      // copia (sim, a instrução MOV serve para copiar) o valor do registrador RSP para RBP
mov    eax,0x9      // coloca o valor literal 9 no registrador EAX
pop    rbp          // retira o valor do topo da pilha e o coloca no registrador RBP
ret                 // simboliza, nesse caso, o fim da função, retomando a execução para onde parou antes da chamada da função main(). Para mais informações a respeito, favor ler os artigos da coluna "Fundamentos para Computação Ofensiva"
```

Para maiores informações sobre a chamada de funções, favor consultar os artigos da coluna "Fundamentos para Computação Ofensiva".

Não é escopo desse artigo discutir os mecanismos que envolvem chamar e sair de uma função, mas por agora o leitor deve saber que o "9" da linha "return 9" é o que aparece na instrução "mov eax,0x9".

No *shell* do *Linux* a variável especial $? contém o código de retorno de um programa. Para então se certificar de que este programa está retornando 9, podemos fazer:

```
$ ./nada
$ echo $?
9
```

Se o resultado for 9 como no exemplo, você fez tudo certo.

E atenção, você já está fazendo engenharia reversa! Olhar um binário compilado e entendê-lo já é pra lá de interessante. No próximo artigo vamos estudar mais sobre identificação de funções, *offsets*, endereços e iniciar o estudo de *patching*, onde modificaremos o binário depois de compilado. Até lá!

[1] http://www.asciitable.com/
[2] https://refspecs.linuxbase.org/elf/elf.pdf
[3] http://github.com/merces/hdump/

FERNANDO MERCÊS

Fernando é Pesquisador de Ameças na Trend Micro. Com foco em segurança de aplicações e sistemas, desenvolve soluções de proteção contra ameaças e ataques, principalmente no Brasil. Tem uma forte ligação com o mundo open source, sendo colaborador da comunidade Debian, desenvolvedor de projetos livres na área de segurança como pev, aleph e T50. Já apresentou pesquisas em eventos como H2HC, FISL e LinuxCon e é um constante estudioso da evolução de ataques cibernéticos.

Stack Frames - O Que São e Para Que Servem? *Parte II*

POR YGOR DA ROCHA PARREIRA

Este artigo está dividido em duas partes onde a primeira parte, publicada na edição número 7 dessa revista, mostrou o funcionamento da *stack* e esta segunda parte mostra o uso dos *stack frames*. As duas partes abusam de exercícios práticos como forma de comprovar os conceitos apresentados.

0x0 – Stack Frames

Sempre que uma função é chamada, cria-se um novo *stack frame*[1] no topo da *stack*. Desta forma mesmo que se faça uso de chamadas recursivas, o conjunto de variáveis locais não interfere com as variáveis locais[2] de outra instância da função. A Figura 1 ilustra como seria o empilhamento dos *stack frames* do programa apresentado na Listagem 1, durante a execução da função quadrado().

```
root@research:~/H2HC_Magazine/Fundamentos/Stack_Frame# cat stack_frame.c
// To Compile: gcc -o stack_frame stack_frame.c
int soma(int, int);
int quadrado(int);
int main()
{
      soma(4, 3); // Primeira Chamada.
      return 0;
}
int soma(int a, int b)
{
      int res;
      res = a + b;
      quadrado(res); // Segunda Chamada.
      return res;
}
int quadrado(int num)
{
      int resultado;
      resultado = num * num;
      return resultado;
}
root@research:~/H2HC_Magazine/Fundamentos/Stack_Frame# gcc -o stack_frame stack_frame.c // Compila o programa.
root@research:~/H2HC_Magazine/Fundamentos/Stack_Frame#
```

Listagem 1 – Fonte e processo de compilação do programa stack_frame, utilizado para demonstrar o empilhamento dos stack frames durante sua execução

Geralmente o registrador EBP é utilizado para identificar uma posição fixa dentro do *stack frame* da função chamada[3], e é conhecido como *stack frame* base *pointer*, ou simplesmente *frame pointer*. Ele permite acesso fácil a parâmetros passados na *stack*, ao *return instruction pointer*, bem como às variáveis locais criadas pelo procedimento.

[1] - *Stack frames são abstrações originadas no hardware e herdadas pela ABI/Sistema Operacional.*
[2]- *Faz-se obvio aqui que estamos falando apenas de variáveis locais que são armazenadas na stack. A forma com que cada tipo de variável (dinâmicas, automáticas, estáticas, etc) são armazenadas na memória não é o escopo principal deste artigo, e será tratado num artigo futuro.*
[3]- *Se compilado com otimização o GCC na versão utilizada neste artigo não faz uso do EBP da forma como explicado aqui.*

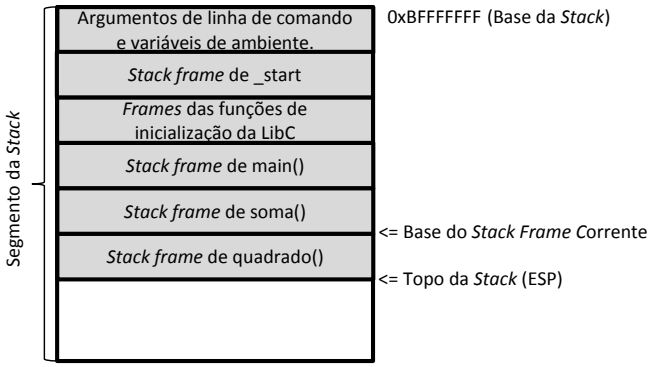


Figura 1 – Empilhamento dos stack frames do programa stack_frame, quando executando a função quadrado

Assim como uma API (*Application Programming Interface*) fornece padronização de interface para o código fonte, a ABI *(Application Binary Interface*) fornece padronização de interface para duas ou mais partes de *software* em uma determinada arquitetura. Ela define como a aplicação deve interagir com ela mesma, como deve interagir com as bibliotecas compartilhadas e até mesmo com o *kernel*. Enquanto a API garante compatibilidade de fontes, a ABI garante compatibilidade binária.

Apesar da IA-32 não requerer alinhamento de *stack*, a ABI do Linux define um alinhamento em fronteiras de *words* (32 *bits* nesta arquitetura)[4]. Também é definido que os argumentos são passados na *stack* em ordem inversa a do protótipo da função (último argumento deve ser inserido na *stack* primeiro), e devem respeitar o alinhamento requerido. Isto significa que, caso necessário, um argumento pode aumentar de tamanho para se tornar múltiplo de *words*. Assim como na IA-32, a ABI do Linux também define que todos os argumentos presentes na *stack* residam no *stack frame* **do chamador**.

A Figura 2 mostra o formato de um *stack frame*. A Listagem 2 mostra a execução passo a passo da função *main*() do programa *stack_frame*, até a chamada da função soma() e criação do novo *stack frame*.

```
root@research:~/H2HC_Magazine/Fundamentos/Stack_
Frame# gdb -q stack_frame
Reading symbols from /root/H2HC_Magazine/Fundamentos/
Stack_Frame/stack_frame...(no debugging symbols
found)...done.
(gdb) disassemble main // Desmontando main() para
encontrar qual o endereço de sua primeira instrução.
Dump of assembler code for function main:
  0x080483dc <+0>:   push  %ebp // Endereço encontrado.
  0x080483dd <+1>:    mov   %esp,%ebp
  0x080483df <+3>:    and   $0xfffffff0,%esp
  0x080483e2 <+6>:    sub   $0x10,%esp
  0x080483e5 <+9>:    movl  $0x3,0x4(%esp)
  0x080483ed <+17>:   movl  $0x4,(%esp)
  0x080483f4 <+24>:   call  0x8048400 <soma>
  0x080483f9 <+29>:   mov   $0x0,%eax
  0x080483fe <+34>:   leave
  0x080483ff <+35>:   ret
End of assembler dump.
```

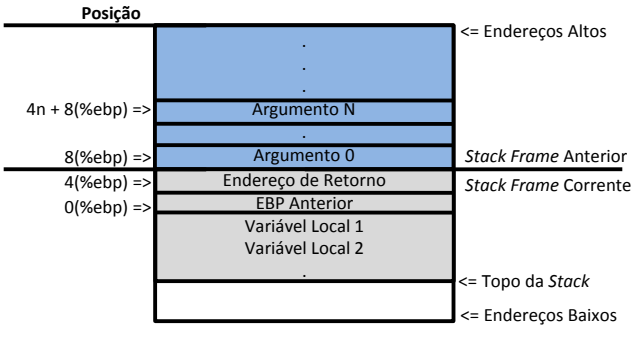


Figura 2 – Composição de um stack frame

[4] *-O tamanho de uma word é controverso em diversas arquiteturas, ou mesmo em uma mesma arquitetura como no caso deste documento, que versa sobre a IA-32 (manual da Intel define como 16 bits, e a ABI do Linux define como 32 bits). Para os fins deste contexto adotamos o tamanho de 32 bits, que é o mesmo definido pela ABI do Linux para a arquitetura em questão.*

```
(gdb) break *0x080483dc // Criando break-point no endereço encontrado.
Breakpoint 1 at 0x80483dc
(gdb) r
Starting program: /root/H2HC_Magazine/Fundamentos/Stack_Frame/stack_frame

Breakpoint 1, 0x080483dc in main ()
(gdb) disassemble main
Dump of assembler code for function main:
=> 0x080483dc <+0>:    push   %ebp // Processo parado aqui (break-point) – indicado pela seta no início da linha.
   0x080483dd <+1>:    mov    %esp,%ebp
   0x080483df <+3>:    and    $0xfffffff0,%esp
   0x080483e2 <+6>:    sub    $0x10,%esp
   0x080483e5 <+9>:    movl   $0x3,0x4(%esp)
   0x080483ed <+17>:   movl   $0x4,(%esp)
   0x080483f4 <+24>:   call   0x8048400 <soma>
   0x080483f9 <+29>:   mov    $0x0,%eax
   0x080483fe <+34>:   leave
   0x080483ff <+35>:   ret
End of assembler dump.
(gdb) i r $ebp $esp // Verifica o conteúdo de EBP e onde (topo da stack) ele será colocado.
ebp            0xbffffc68      0xbffffc68
esp            0xbffffbec      0xbffffbec
(gdb) x/x $esp // Verifica o endereço de retorno de main().
0xbffffbec:    0xb7e8ce46 // Endereço de retorno de main(), endereço dentro do mapeamento da LibC.
(gdb) x/4i __libc_start_main+227 // Verifica para onde main() retorna[5].
   0xb7e8ce43 <__libc_start_main+227>:  call   *0x8(%ebp) // Aqui faz sentido o endereço de main() ser o primeiro
parâmetro – último argumento a ser colocado na stack antes do call – da rotina de inicialização da LibC.
   0xb7e8ce46 <__libc_start_main+230>:  mov    %eax,(%esp) // Retorna pra cá.
   0xb7e8ce49 <__libc_start_main+233>:  call   0xb7ea5550 <exit>
   0xb7e8ce4e <__libc_start_main+238>:  xor    %ecx,%ecx
(gdb) ni
0x080483dd in main ()
(gdb) disassemble main
Dump of assembler code for function main:
   0x080483dc <+0>:    push   %ebp // Início do prólogo de main().
=> 0x080483dd <+1>:    mov    %esp,%ebp // Execução parada aqui – Fim do prólogo.
   0x080483df <+3>:    and    $0xfffffff0,%esp
   0x080483e2 <+6>:    sub    $0x10,%esp
   0x080483e5 <+9>:    movl   $0x3,0x4(%esp)
   0x080483ed <+17>:   movl   $0x4,(%esp)
   0x080483f4 <+24>:   call   0x8048400 <soma>
   0x080483f9 <+29>:   mov    $0x0,%eax
   0x080483fe <+34>:   leave
   0x080483ff <+35>:   ret
End of assembler dump.
(gdb) i r $ebp $esp
ebp            0xbffffc68      0xbffffc68
esp            0xbffffbe8      0xbffffbe8
(gdb) x/x $esp
0xbffffbe8:    0xbffffc68 // Conteúdo do topo da stack (antigo frame pointer).
(gdb) ni
0x080483df in main ()
(gdb) disassemble main
```

[5] - *Ver a terceira questão da sessão 0x2 – Perguntas Mais Frequentes.*

```
Dump of assembler code for function main:
  0x080483dc <+0>:    push   %ebp
  0x080483dd <+1>:    mov    %esp,%ebp
=> 0x080483df <+3>:    and    $0xfffffff0,%esp // Alinhando a stack em fronteiras de 16 bytes[6].
  0x080483e2 <+6>:    sub    $0x10,%esp
  0x080483e5 <+9>:    movl   $0x3,0x4(%esp)
  0x080483ed <+17>:   movl   $0x4,(%esp)
  0x080483f4 <+24>:   call   0x8048400 <soma>
  0x080483f9 <+29>:   mov    $0x0,%eax
  0x080483fe <+34>:   leave
  0x080483ff <+35>:   ret
End of assembler dump.
(gdb) i r $ebp $esp
ebp            0xbffffbe8      0xbffffbe8 // Pronto, EBP já contém o frame pointer do stack frame corrente.
esp            0xbffffbe8      0xbffffbe8
(gdb) ni
0x080483e2 in main ()
(gdb) disassemble main
Dump of assembler code for function main:
  0x080483dc <+0>:    push   %ebp
  0x080483dd <+1>:    mov    %esp,%ebp
  0x080483df <+3>:    and    $0xfffffff0,%esp
=> 0x080483e2 <+6>:    sub    $0x10,%esp // Aloca espaço na stack.
  0x080483e5 <+9>:    movl   $0x3,0x4(%esp)
  0x080483ed <+17>:   movl   $0x4,(%esp)
  0x080483f4 <+24>:   call   0x8048400 <soma>
  0x080483f9 <+29>:   mov    $0x0,%eax
  0x080483fe <+34>:   leave
  0x080483ff <+35>:   ret
End of assembler dump.
(gdb) i r $esp
esp            0xbffffbe0      0xbffffbe0 // Stack alinhada.
(gdb) ni
0x080483e5 in main ()
(gdb) disassemble main
Dump of assembler code for function main:
  0x080483dc <+0>:    push   %ebp
  0x080483dd <+1>:    mov    %esp,%ebp
  0x080483df <+3>:    and    $0xfffffff0,%esp
  0x080483e2 <+6>:    sub    $0x10,%esp
=> 0x080483e5 <+9>:    movl   $0x3,0x4(%esp) // Passa o segundo parâmetro da chamada soma(4, 3), em ordem
inversa conforme calling convention definida pela ABI.
  0x080483ed <+17>:   movl   $0x4,(%esp)
  0x080483f4 <+24>:   call   0x8048400 <soma>
  0x080483f9 <+29>:   mov    $0x0,%eax
  0x080483fe <+34>:   leave
  0x080483ff <+35>:   ret
End of assembler dump.
(gdb) i r $esp
esp            0xbffffbd0      0xbffffbd0 // Espaço alocado na stack.
 (gdb) ni
0x080483ed in main ()
(gdb) disassemble main
```

[6] - *O motivo disto é explicado posteriormente.*

```
Dump of assembler code for function main:
  0x080483dc <+0>:    push   %ebp
  0x080483dd <+1>:    mov    %esp,%ebp
  0x080483df <+3>:    and    $0xfffffff0,%esp
  0x080483e2 <+6>:    sub    $0x10,%esp
  0x080483e5 <+9>:    movl   $0x3,0x4(%esp)
=> 0x080483ed <+17>:   movl   $0x4,(%esp) // Passa o primeiro parâmetro da chamada soma(4, 3).
  0x080483f4 <+24>:   call   0x8048400 <soma>
  0x080483f9 <+29>:   mov    $0x0,%eax
  0x080483fe <+34>:   leave
  0x080483ff <+35>:   ret
End of assembler dump.
(gdb) x/x $esp+4
0xbffffbd4:    0x00000003 // Segundo parâmetro da chamada soma(4, 3) já em seu lugar.
(gdb) ni
0x080483f4 in main ()
(gdb) disassemble main
Dump of assembler code for function main:
  0x080483dc <+0>:    push   %ebp
  0x080483dd <+1>:    mov    %esp,%ebp
  0x080483df <+3>:    and    $0xfffffff0,%esp
  0x080483e2 <+6>:    sub    $0x10,%esp
  0x080483e5 <+9>:    movl   $0x3,0x4(%esp)
  0x080483ed <+17>:   movl   $0x4,(%esp)
=> 0x080483f4 <+24>:   call   0x8048400 <soma> // Chama a função soma(), criando um novo stack frame. Esta
instrução coloca o endereço da próxima instrução (0x080483f9) no topo da stack, e salta para o endereço referenciado
(0x8048400).
  0x080483f9 <+29>:   mov    $0x0,%eax
  0x080483fe <+34>:   leave
  0x080483ff <+35>:   ret
End of assembler dump.
(gdb) x/x $esp
0xbffffbd0:    0x00000004 // Primeiro parâmetro da chamada soma(4, 3) já em seu lugar.
(gdb) stepi // Entra na função chamada (soma()).
0x08048400 in soma ()
(gdb) disassemble soma
Dump of assembler code for function soma:
=> 0x08048400 <+0>:    push   %ebp // Início do prólogo de soma().
  0x08048401 <+1>:    mov    %esp,%ebp
  0x08048403 <+3>:    sub    $0x28,%esp
  0x08048406 <+6>:    mov    0xc(%ebp),%eax
  0x08048409 <+9>:    mov    0x8(%ebp),%edx
  0x0804840c <+12>:   add    %edx,%eax
  0x0804840e <+14>:   mov    %eax,-0xc(%ebp)
  0x08048411 <+17>:   mov    -0xc(%ebp),%eax
  0x08048414 <+20>:   mov    %eax,(%esp)
  0x08048417 <+23>:   call   0x8048421 <quadrado>
  0x0804841c <+28>:   mov    -0xc(%ebp),%eax
  0x0804841f <+31>:   leave
  0x08048420 <+32>:   ret
End of assembler dump.
(gdb) i r $esp
esp            0xbffffbcc      0xbffffbcc
(gdb) x/x $esp // Verificando o endereço de retorno de soma().
0xbffffbcc:    0x080483f9 // Endereço de retorno de soma().
(gdb)
```

Listagem 2 – Execução passo a passo da função main() do programa stack_frame, até a criação do novo stack frame

Durante a sessão de execução passo a passo apresentada na Listagem 2, observa-se que main() alinha a *stack* em fronteiras de 16 *bytes*, e não de 4 *bytes* como requerido pela ABI do Linux para esta arquitetura. Mas porque isto? Apesar da IA-32 ser descrita como uma arquitetura do tipo *soft alignment*, ou seja, que não exige alinhamento de memória, ela possui uma extensão do seu conjunto de instruções onde a maioria destas exigem o alinhamento em fronteiras de 16 *bytes*. Exemplos destas instruções podem ser encontrados em extensões SIMD (*Single Instruction, Multiple Data*), que na Intel recebeu nomes de SSE, SSE2, SSE3, SSSE3 e SSE4[7].

O GCC utiliza a opção de compilação "-mpreferred-stack-boundary=**num**" para indicar qual é o fator de alinhamento da *stack*, onde **num** é o expoente de base dois deste fator. Como o padrão é **num = 4**, $2^4 = 16$ [8]. Até mesmo a alocação de memória subsequente desta função (instrução no endereço 0x080483e2) continua respeitando este alinhamento.

A instrução **call** cria um novo *stack frame* e salva no topo da *stack* o endereço da instrução subsequente a ela, de forma que o programa consiga prosseguir sua execução quando a função chamada terminar. Normalmente em todo início de função é executado o que é conhecido como prólogo[9], que é composto das seguintes instruções:

"push %ebp
mov %esp, %ebp"

Estas instruções servem para salvar o *frame pointer* do *stack frame* anterior, e configurar o *frame pointer* para o *stack frame* corrente.

A Listagem 2 também mostra que o endereço de retorno salvo para quando main() encerrar sua execução aponta para código dentro da rotina de inicialização da LibC (instrução __libc_start_main+230), o que corrobora a informação de que as rotinas de inicialização da LibC são executadas antes de main()[10]. Também nota-se a conformidade da passagem dos parâmetros com as especificações da ABI, ou seja, são passados para a *stack* em ordem inversa da especificada na função em C[11].

A Figura 3 mostra passo a passo o que ocorre na *stack* e em alguns registradores da CPU, durante a execução da função main() do programa stack_frame. A Figura 4 mostra como ficou a organização dos *stack frames*, após a chamada da função soma() no programa stack_frame.

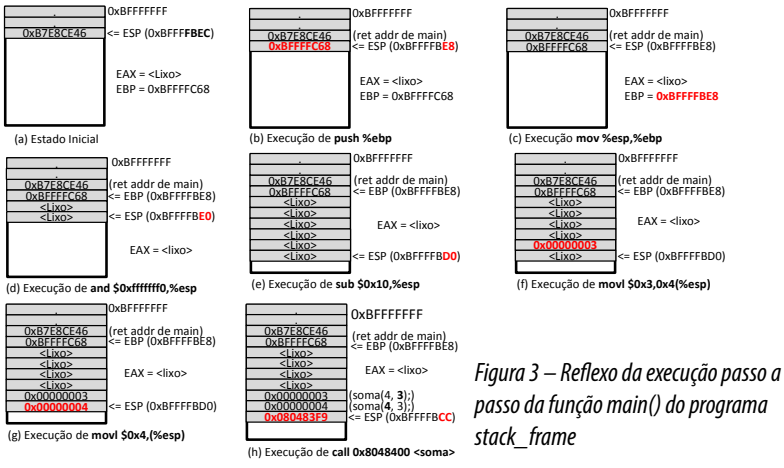


Figura 3 – Reflexo da execução passo a passo da função main() do programa stack_frame

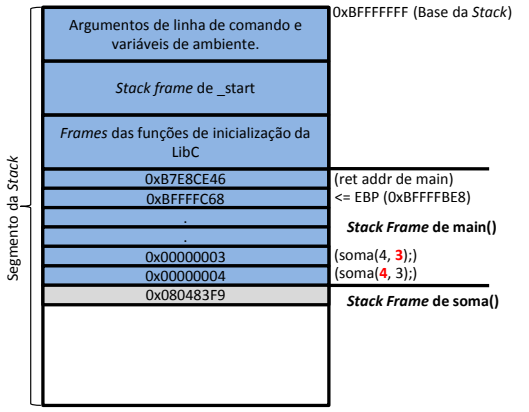


Figura 4 – Organização dos stack frames do programa stack_frame, logo após a chamada da função soma()

7 - *Manual da Intel, Volume 1: Basic Architecture.*
8 - *Este valor padrão pode variar entre versões diferentes do gcc.*
9 - *Isso pode variar de acordo com a calling convention e arquiteturas adotadas. Algumas literaturas também consideram a alocação de espaço para variáveis locais como parte do prólogo.*
10 - *Ver a primeira parte do artigo, na edição n. 7 da H2HC Magazine.*
11 - *Ver a primeira questão da sessão 0x2 – Perguntas Mais Frequentes.*

A Listagem 3 mostra a execução passo a passo da função soma() no programa stack_frame.

```
(gdb) disassemble soma
Dump of assembler code for function soma:
=> 0x08048400 <+0>:    push   %ebp // Prólogo de soma().
   0x08048401 <+1>:    mov    %esp,%ebp // Fim do prólogo de soma().
   0x08048403 <+3>:    sub    $0x28,%esp
   0x08048406 <+6>:    mov    0xc(%ebp),%eax
   0x08048409 <+9>:    mov    0x8(%ebp),%edx
   0x0804840c <+12>:   add    %edx,%eax
   0x0804840e <+14>:   mov    %eax,-0xc(%ebp)
   0x08048411 <+17>:   mov    -0xc(%ebp),%eax
   0x08048414 <+20>:   mov    %eax,(%esp)
   0x08048417 <+23>:   call   0x8048421 <quadrado>
   0x0804841c <+28>:   mov    -0xc(%ebp),%eax
   0x0804841f <+31>:   leave
   0x08048420 <+32>:   ret
End of assembler dump.
(gdb) i r $esp $ebp
esp            0xbffffbcc      0xbffffbcc
ebp            0xbffffbe8      0xbffffbe8
(gdb) x/x $esp // Verificando o endereço de retorno de soma().
0xbffffbcc:    0x080483f9 // Endereço de retorno de soma().
(gdb) ni
0x08048401 in soma ()
(gdb) disassemble soma
Dump of assembler code for function soma:
   0x08048400 <+0>:    push   %ebp
=> 0x08048401 <+1>:    mov    %esp,%ebp
   0x08048403 <+3>:    sub    $0x28,%esp
   0x08048406 <+6>:    mov    0xc(%ebp),%eax
   0x08048409 <+9>:    mov    0x8(%ebp),%edx
   0x0804840c <+12>:   add    %edx,%eax
   0x0804840e <+14>:   mov    %eax,-0xc(%ebp)
   0x08048411 <+17>:   mov    -0xc(%ebp),%eax
   0x08048414 <+20>:   mov    %eax,(%esp)
   0x08048417 <+23>:   call   0x8048421 <quadrado>
   0x0804841c <+28>:   mov    -0xc(%ebp),%eax
   0x0804841f <+31>:   leave
   0x08048420 <+32>:   ret
End of assembler dump.
(gdb) x/x $esp
0xbffffbc8:    0xbffffbe8 // O frame pointer do stack frame anterior (relativo a função main()) já está no topo da
stack – primeiro passo do prólogo.
(gdb) ni
0x08048403 in soma ()
(gdb) disassemble soma
Dump of assembler code for function soma:
   0x08048400 <+0>:    push   %ebp
   0x08048401 <+1>:    mov    %esp,%ebp
=> 0x08048403 <+3>:    sub    $0x28,%esp // Aloca memória para variáveis locais.
   0x08048406 <+6>:    mov    0xc(%ebp),%eax
   0x08048409 <+9>:    mov    0x8(%ebp),%edx
```

```
   0x0804840c <+12>:    add    %edx,%eax
   0x0804840e <+14>:    mov    %eax,-0xc(%ebp)
   0x08048411 <+17>:    mov    -0xc(%ebp),%eax
   0x08048414 <+20>:    mov    %eax,(%esp)
   0x08048417 <+23>:    call   0x8048421 <quadrado>
   0x0804841c <+28>:    mov    -0xc(%ebp),%eax
   0x0804841f <+31>:    leave
   0x08048420 <+32>:    ret
End of assembler dump.
(gdb) i r $ebp $esp
ebp            0xbffffbc8      0xbffffbc8 // Frame pointer ajustado para o stack frame corrente (relativo a função soma()) – último passo do prólogo.
esp            0xbffffbc8      0xbffffbc8
(gdb) ni
0x08048406 in soma ()
(gdb) disassemble soma
Dump of assembler code for function soma:
   0x08048400 <+0>:     push   %ebp
   0x08048401 <+1>:     mov    %esp,%ebp
   0x08048403 <+3>:     sub    $0x28,%esp
=> 0x08048406 <+6>:     mov    0xc(%ebp),%eax // Copia o segundo argumento de soma(4, 3) para dentro do registrador EAX.
   0x08048409 <+9>:     mov    0x8(%ebp),%edx
   0x0804840c <+12>:    add    %edx,%eax
   0x0804840e <+14>:    mov    %eax,-0xc(%ebp)
   0x08048411 <+17>:    mov    -0xc(%ebp),%eax
   0x08048414 <+20>:    mov    %eax,(%esp)
   0x08048417 <+23>:    call   0x8048421 <quadrado>
   0x0804841c <+28>:    mov    -0xc(%ebp),%eax
   0x0804841f <+31>:    leave
   0x08048420 <+32>:    ret
End of assembler dump.
(gdb) i r $esp
esp            0xbffffba0      0xbffffba0 // Memória alocada para variáveis locais.
(gdb) x/x $ebp+0xc
0xbffffbd4:    0x00000003 // Segundo parâmetro de soma(4, 3).
(gdb) i r $eax
eax            0xbffffc94      -1073742700 // Lixo contido em EAX.
(gdb) ni
0x08048409 in soma ()
(gdb) disassemble soma
Dump of assembler code for function soma:
   0x08048400 <+0>:     push   %ebp
   0x08048401 <+1>:     mov    %esp,%ebp
   0x08048403 <+3>:     sub    $0x28,%esp
   0x08048406 <+6>:     mov    0xc(%ebp),%eax
=> 0x08048409 <+9>:     mov    0x8(%ebp),%edx // Copia o primeiro argumento de soma(4, 3) para dentro do registrador EDX.
   0x0804840c <+12>:    add    %edx,%eax
   0x0804840e <+14>:    mov    %eax,-0xc(%ebp)
   0x08048411 <+17>:    mov    -0xc(%ebp),%eax
   0x08048414 <+20>:    mov    %eax,(%esp)
   0x08048417 <+23>:    call   0x8048421 <quadrado>
   0x0804841c <+28>:    mov    -0xc(%ebp),%eax
   0x0804841f <+31>:    leave
```

```
  0x08048420 <+32>:    ret
End of assembler dump.
(gdb) i r $eax
eax            0x3     3 // Segundo argumento já em EAX.
(gdb) x/x $ebp+0x8
0xbffffbd0:    0x00000004 // Primeiro argumento de soma(4, 3).
(gdb) i r $edx
edx            0x1     1 // Lixo contido em EDX.
(gdb) ni
0x0804840c in soma ()
(gdb) disassemble soma
Dump of assembler code for function soma:
  0x08048400 <+0>:    push   %ebp
  0x08048401 <+1>:    mov    %esp,%ebp
  0x08048403 <+3>:    sub    $0x28,%esp
  0x08048406 <+6>:    mov    0xc(%ebp),%eax
  0x08048409 <+9>:    mov    0x8(%ebp),%edx
=> 0x0804840c <+12>:    add    %edx,%eax // Faz a soma dos dois argumentos passados para a função, e armazena
o resultado em EAX.
  0x0804840e <+14>:    mov    %eax,-0xc(%ebp)
  0x08048411 <+17>:    mov    -0xc(%ebp),%eax
  0x08048414 <+20>:    mov    %eax,(%esp)
  0x08048417 <+23>:    call   0x8048421 <quadrado>
  0x0804841c <+28>:    mov    -0xc(%ebp),%eax
  0x0804841f <+31>:    leave
  0x08048420 <+32>:    ret
End of assembler dump.
(gdb) i r $edx $eax
edx            0x4     4 // Primeiro argumento em EDX.
eax            0x3     3 // Segundo argumento em EAX.
(gdb) ni
0x0804840e in soma ()
(gdb) disassemble soma
Dump of assembler code for function soma:
  0x08048400 <+0>:    push   %ebp
  0x08048401 <+1>:    mov    %esp,%ebp
  0x08048403 <+3>:    sub    $0x28,%esp
  0x08048406 <+6>:    mov    0xc(%ebp),%eax
  0x08048409 <+9>:    mov    0x8(%ebp),%edx
  0x0804840c <+12>:    add    %edx,%eax
=> 0x0804840e <+14>:    mov    %eax,-0xc(%ebp) // Copia o resultado da soma para a variável local res.
  0x08048411 <+17>:    mov    -0xc(%ebp),%eax
  0x08048414 <+20>:    mov    %eax,(%esp)
  0x08048417 <+23>:    call   0x8048421 <quadrado>
  0x0804841c <+28>:    mov    -0xc(%ebp),%eax
  0x0804841f <+31>:    leave
  0x08048420 <+32>:    ret
End of assembler dump.
(gdb) i r $eax
eax            0x7     7 // Resultado da soma armazenado em EAX.
(gdb) x/x $ebp-0xc
0xbffffbbc:    0x0804849b // Lixo contido na posição da variável local res, da função soma().
(gdb) ni
0x08048411 in soma ()
(gdb) disassemble soma
```

```
Dump of assembler code for function soma:
   0x08048400 <+0>:    push   %ebp
   0x08048401 <+1>:    mov    %esp,%ebp
   0x08048403 <+3>:    sub    $0x28,%esp
   0x08048406 <+6>:    mov    0xc(%ebp),%eax
   0x08048409 <+9>:    mov    0x8(%ebp),%edx
   0x0804840c <+12>:   add    %edx,%eax
   0x0804840e <+14>:   mov    %eax,-0xc(%ebp)
=> 0x08048411 <+17>:   mov    -0xc(%ebp),%eax // Copia o conteúdo da variável res para o registrador EAX.
   0x08048414 <+20>:   mov    %eax,(%esp)
   0x08048417 <+23>:   call   0x8048421 <quadrado>
   0x0804841c <+28>:   mov    -0xc(%ebp),%eax
   0x0804841f <+31>:   leave
   0x08048420 <+32>:   ret
End of assembler dump.
(gdb) x/x $ebp-0xc
0xbffffbbc:    0x00000007 // Resultado da soma armazenado na variável local res.
(gdb) ni
0x08048414 in soma ()
(gdb) disassemble soma
Dump of assembler code for function soma:
   0x08048400 <+0>:    push   %ebp
   0x08048401 <+1>:    mov    %esp,%ebp
   0x08048403 <+3>:    sub    $0x28,%esp
   0x08048406 <+6>:    mov    0xc(%ebp),%eax
   0x08048409 <+9>:    mov    0x8(%ebp),%edx
   0x0804840c <+12>:   add    %edx,%eax
   0x0804840e <+14>:   mov    %eax,-0xc(%ebp)
   0x08048411 <+17>:   mov    -0xc(%ebp),%eax
=> 0x08048414 <+20>:   mov    %eax,(%esp) // Passa o resultado da soma como argumento para a função quadrado().
   0x08048417 <+23>:   call   0x8048421 <quadrado>
   0x0804841c <+28>:   mov    -0xc(%ebp),%eax
   0x0804841f <+31>:   leave
   0x08048420 <+32>:   ret
End of assembler dump.
(gdb) i r $eax
eax            0x7      7 // Conteúdo da variável local res, armazenado em EAX.
(gdb) x/x $esp
0xbffffba0:    0x00000000 // Vendo o lixo que se encontra no topo da stack.
(gdb) ni
0x08048417 in soma ()
(gdb) disassemble soma
Dump of assembler code for function soma:
   0x08048400 <+0>:    push   %ebp
   0x08048401 <+1>:    mov    %esp,%ebp
   0x08048403 <+3>:    sub    $0x28,%esp
   0x08048406 <+6>:    mov    0xc(%ebp),%eax
   0x08048409 <+9>:    mov    0x8(%ebp),%edx
   0x0804840c <+12>:   add    %edx,%eax
   0x0804840e <+14>:   mov    %eax,-0xc(%ebp)
   0x08048411 <+17>:   mov    -0xc(%ebp),%eax
   0x08048414 <+20>:   mov    %eax,(%esp)
=> 0x08048417 <+23>:    call   0x8048421 <quadrado> // Chama a função quadrado(), criando um novo stack
frame. Esta instrução coloca o endereço da próxima instrução (0x0804841c) no topo da stack, e salta para o endereço
referenciado (0x8048421).
```

```
  0x0804841c <+28>:    mov    -0xc(%ebp),%eax
  0x0804841f <+31>:    leave
  0x08048420 <+32>:    ret
End of assembler dump.
(gdb) x/x $esp
0xbffffba0:    0x00000007 // Argumento para a função quadrado() passado.
(gdb) i r $esp
esp            0xbffffba0      0xbffffba0 // Visualizando para onde o stack pointer está apontando (qual o endereço do topo da stack).
(gdb) stepi
0x08048421 in quadrado ()
(gdb) disassemble quadrado
Dump of assembler code for function quadrado:
=> 0x08048421 <+0>:    push   %ebp
  0x08048422 <+1>:    mov    %esp,%ebp
0x08048424 <+3>:    sub    $0x10,%esp
  0x08048427 <+6>:    mov    0x8(%ebp),%eax
  0x0804842a <+9>:    imul   0x8(%ebp),%eax
  0x0804842e <+13>:    mov    %eax,-0x4(%ebp)
  0x08048431 <+16>:    mov    -0x4(%ebp),%eax
  0x08048434 <+19>:    leave
  0x08048435 <+20>:    ret
End of assembler dump.
(gdb) x/x $esp
0xbffffb9c:    0x0804841c // Stack pointer decrementado, e endereço de retorno da função quadrado() armazenado no novo stack frame.
(gdb)
```

Listagem 3 – Execução passo a passo da função soma() do programa stack_frame, até a criação do novo stack frame

A execução passo a passo da função soma() mostra que os parâmetros são copiados para os registradores de uso geral EAX e EDX, somados, e depois o resultado é armazenado na posição de memória alocada para a variável local **res**. A passagem do parâmetro para a função quadrado() ocorre com duas instruções: Primeiro copia-se o conteúdo da variável **res** para o registrador EAX, e em seguida copia-se o conteúdo de EAX para o topo da stack. Mas porque não copiar direto entre as posições de memória? Porque a IA-32 não permite cópia direta de uma posição de memória para outra. Finalizado as operações da função soma() e a passagem do argumento, a instrução **call** cria um novo *stack frame* e salva no topo da *stack* o endereço da instrução subsequente.

A Figura 5 mostra passo a passo o que ocorre na *stack* e em alguns registradores da CPU durante a execução da função soma() no programa stack_frame. A Figura 6 mostra como ficou a organização dos *stack frames*, após a chamada da função quadrado() no programa stack_frame.

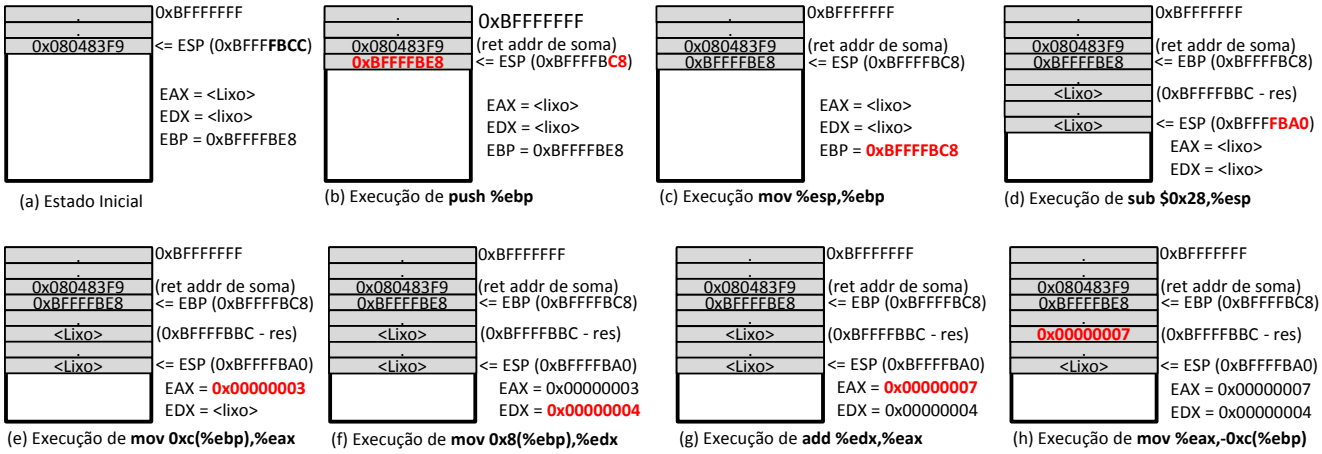


Figura 5 – Reflexo da execução passo a passo da função soma() no programa stack_frame

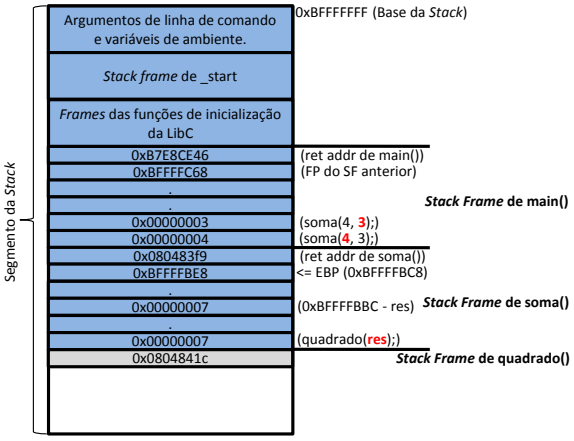


Figura 6 – Organização dos stack frames do programa stack_frame, logo após a chamada da função quadrado()

A Listagem 4 mostra a execução passo a passo da função quadrado().

```
(gdb) disassemble quadrado
Dump of assembler code for function quadrado:
=> 0x08048421 <+0>:    push   %ebp
   0x08048422 <+1>:    mov    %esp,%ebp
   0x08048424 <+3>:    sub    $0x10,%esp
   0x08048427 <+6>:    mov    0x8(%ebp),%eax
   0x0804842a <+9>:    imul   0x8(%ebp),%eax
   0x0804842e <+13>:   mov    %eax,-0x4(%ebp)
   0x08048431 <+16>:   mov    -0x4(%ebp),%eax
   0x08048434 <+19>:   leave
   0x08048435 <+20>:   ret
End of assembler dump.
(gdb) i r $esp $ebp
esp            0xbffffb9c      0xbffffb9c
ebp            0xbffffbc8      0xbffffbc8
(gdb) x/x $esp
0xbffffb9c:    0x0804841c // Endereço de retorno de quadrado().
(gdb) ni
0x08048422 in quadrado ()
(gdb) disassemble quadrado
Dump of assembler code for function quadrado:
   0x08048421 <+0>:    push   %ebp
```

```
=> 0x08048422 <+1>:    mov    %esp,%ebp
   0x08048424 <+3>:    sub    $0x10,%esp
   0x08048427 <+6>:    mov    0x8(%ebp),%eax
   0x0804842a <+9>:    imul   0x8(%ebp),%eax
   0x0804842e <+13>:   mov    %eax,-0x4(%ebp)
   0x08048431 <+16>:   mov    -0x4(%ebp),%eax
   0x08048434 <+19>:   leave
   0x08048435 <+20>:   ret
End of assembler dump.
(gdb) x/x $esp
0xbffffb98:    0xbffffbc8 // Frame pointer do stack frame anterior, salvo no topo da stack no stack frame corrente.
(gdb) ni
0x08048424 in quadrado ()
(gdb) disassemble quadrado
Dump of assembler code for function quadrado:
   0x08048421 <+0>:    push   %ebp
   0x08048422 <+1>:    mov    %esp,%ebp
=> 0x08048424 <+3>:    sub    $0x10,%esp
   0x08048427 <+6>:    mov    0x8(%ebp),%eax
   0x0804842a <+9>:    imul   0x8(%ebp),%eax
   0x0804842e <+13>:   mov    %eax,-0x4(%ebp)
   0x08048431 <+16>:   mov    -0x4(%ebp),%eax
   0x08048434 <+19>:   leave
   0x08048435 <+20>:   ret
End of assembler dump.
(gdb) i r $esp $ebp
esp            0xbffffb98      0xbffffb98
ebp            0xbffffb98      0xbffffb98 // Frame pointer ajustado para o stack frame corrente.
(gdb) ni
0x08048427 in quadrado ()
(gdb) disassemble quadrado
Dump of assembler code for function quadrado:
   0x08048421 <+0>:    push   %ebp
   0x08048422 <+1>:    mov    %esp,%ebp
   0x08048424 <+3>:    sub    $0x10,%esp
=> 0x08048427 <+6>:    mov    0x8(%ebp),%eax // Copia o argumento passado para o registrador EAX.
   0x0804842a <+9>:    imul   0x8(%ebp),%eax
   0x0804842e <+13>:   mov    %eax,-0x4(%ebp)
   0x08048431 <+16>:   mov    -0x4(%ebp),%eax
   0x08048434 <+19>:   leave
   0x08048435 <+20>:   ret
End of assembler dump.
(gdb) i r $esp $eax
esp            0xbffffb88      0xbffffb88 // Espaço para variáveis locais alocado com sucesso.
eax            0x7     7 // Lixo contido em EAX.
(gdb) x/x $ebp+8
0xbffffba0:    0x00000007 // Argumento passado no stack frame anterior.
(gdb) ni
0x0804842a in quadrado ()
(gdb) disassemble quadrado
Dump of assembler code for function quadrado:
   0x08048421 <+0>:    push   %ebp
   0x08048422 <+1>:    mov    %esp,%ebp
   0x08048424 <+3>:    sub    $0x10,%esp
   0x08048427 <+6>:    mov    0x8(%ebp),%eax
```

```
=> 0x0804842a <+9>:    imul   0x8(%ebp),%eax // Multiplica o argumento do tipo inteiro passado no stack frame anterior com sua cópia contida em EAX, e armazena o resultado em EAX.
   0x0804842e <+13>:   mov    %eax,-0x4(%ebp)
   0x08048431 <+16>:   mov    -0x4(%ebp),%eax
   0x08048434 <+19>:   leave
   0x08048435 <+20>:   ret
End of assembler dump.
(gdb) i r $eax
eax            0x7      7 // Argumento armazenado em EAX.
(gdb) ni
0x0804842e in quadrado ()
(gdb) disassemble quadrado
Dump of assembler code for function quadrado:
   0x08048421 <+0>:    push   %ebp
   0x08048422 <+1>:    mov    %esp,%ebp
   0x08048424 <+3>:    sub    $0x10,%esp
   0x08048427 <+6>:    mov    0x8(%ebp),%eax
   0x0804842a <+9>:    imul   0x8(%ebp),%eax
=> 0x0804842e <+13>:    mov    %eax,-0x4(%ebp) // Armazena o resultado da multiplicação contido em EAX no espaço reservado para a variável local resultado.
   0x08048431 <+16>:   mov    -0x4(%ebp),%eax
   0x08048434 <+19>:   leave
   0x08048435 <+20>:   ret
End of assembler dump.
(gdb) i r $eax
eax            0x31     49 // Resultado da multiplicação de inteiros.
(gdb) x/x $ebp-0x4
0xbffffb94:    0xb7ea55f5 // Lixo contido no espaço reservado para a variável local resultado.
(gdb) ni
0x08048431 in quadrado ()
(gdb) disassemble quadrado
Dump of assembler code for function quadrado:
   0x08048421 <+0>:    push   %ebp
   0x08048422 <+1>:    mov    %esp,%ebp
   0x08048424 <+3>:    sub    $0x10,%esp
   0x08048427 <+6>:    mov    0x8(%ebp),%eax
   0x0804842a <+9>:    imul   0x8(%ebp),%eax
   0x0804842e <+13>:   mov    %eax,-0x4(%ebp)
=> 0x08048431 <+16>:   mov    -0x4(%ebp),%eax // Coloca o resultado da multiplicação em EAX. A ABI define que os retornos de funções devem estar contidos no registrador EAX.
   0x08048434 <+19>:   leave
   0x08048435 <+20>:   ret
End of assembler dump.
(gdb) x/x $ebp-0x4
0xbffffb94:    0x00000031 // Resultado armazenado no espaço reservado para a variável local resultado.
(gdb) ni
0x08048434 in quadrado ()
(gdb) disassemble quadrado
Dump of assembler code for function quadrado:
   0x08048421 <+0>:    push   %ebp
   0x08048422 <+1>:    mov    %esp,%ebp
   0x08048424 <+3>:    sub    $0x10,%esp
   0x08048427 <+6>:    mov    0x8(%ebp),%eax
   0x0804842a <+9>:    imul   0x8(%ebp),%eax
   0x0804842e <+13>:   mov    %eax,-0x4(%ebp)
```

```
  0x08048431 <+16>:   mov   -0x4(%ebp),%eax
=> 0x08048434 <+19>:   leave     0x08048435 <+20>:   ret
End of assembler dump.
(gdb) i r $eax
eax            0x31    49 // Valor retornado pela função quadrado() colocado em seu devido lugar (registrador EAX) antes do epílogo.
(gdb)
```

Listagem 4 – Execução passo a passo da função quadrado() no programa stack_frame, até antes da execução do epílogo

É interessante notar que durante a execução das funções o *frame pointer* sempre aponta para um ponto fixo dentro do *stack frame* corrente, onde tanto os parâmetros passados como as variáveis locais são referenciadas dando seu deslocamento em relação ao EBP. Ou seja, quase sempre EBP + Deslocamento referencia argumentos e EBP - Deslocamento referencia variáveis locais. Adicionalmente, a ABI define que valores de retorno de funções devem estar contidos no registrador EAX.

A Figura 7 mostra passo a passo o que ocorre na *stack* e em alguns registradores da CPU, durante a execução da função quadrado() no programa stack_frame. A Figura 8 mostra como ficou a organização dos *stack frames*, até antes das instruções de epílogo da função quadrado().

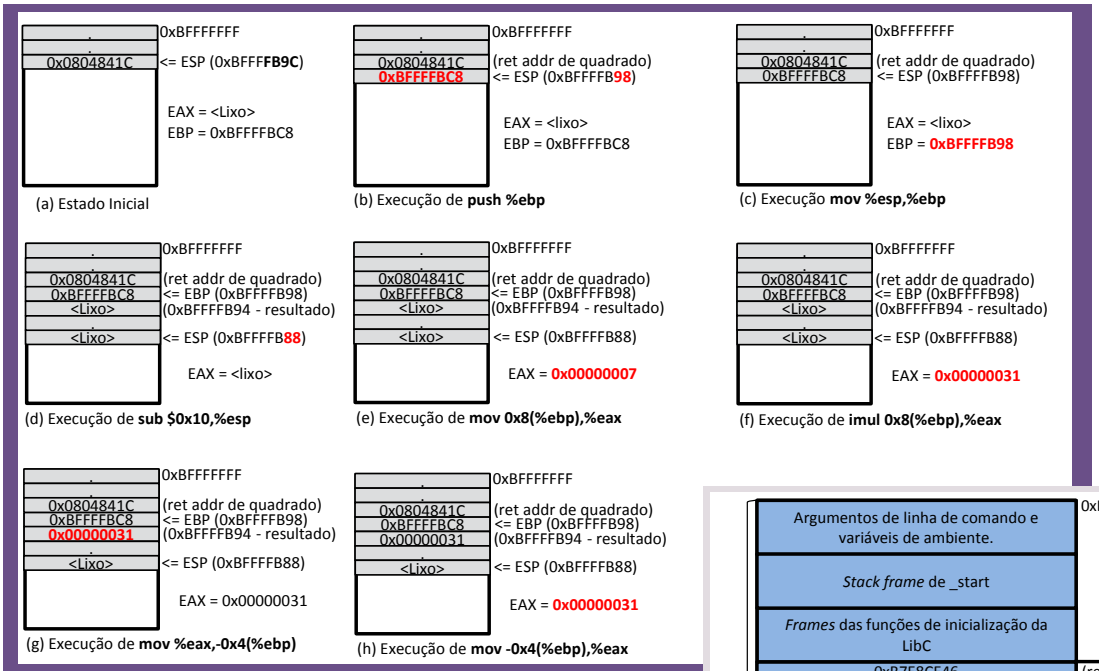


Figura 7 – Reflexo da execução passo a passo da função quadrado() do programa stack_frame

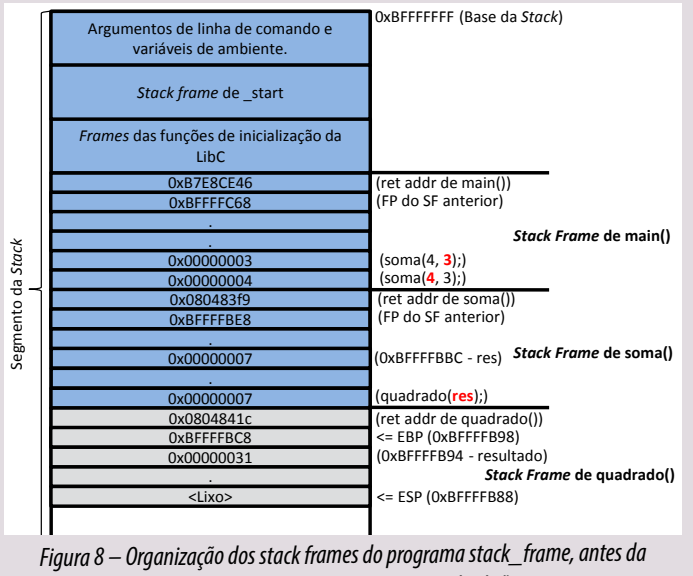


Figura 8 – Organização dos stack frames do programa stack_frame, antes da execução do epílogo da função quadrado()

A sequência final das duas instruções nas funções apresentadas é conhecida como epílogo e serve para desalocar o *stack frame* corrente, recuperar o frame pointer do *stack frame* do chamador e continuar a execução de suas instruções no ponto subsequente que ocorreu a chamada da função/procedimento recém terminado.

A Listagem 5 mostra a execução passo a passo do epílogo da função quadrado().

```
(gdb) disassemble quadrado
Dump of assembler code for function quadrado:
  0x08048421 <+0>:    push   %ebp
  0x08048422 <+1>:    mov    %esp,%ebp
  0x08048424 <+3>:    sub    $0x10,%esp
  0x08048427 <+6>:    mov    0x8(%ebp),%eax
  0x0804842a <+9>:    imul   0x8(%ebp),%eax
  0x0804842e <+13>:   mov    %eax,-0x4(%ebp)
  0x08048431 <+16>:   mov    -0x4(%ebp),%eax
=> 0x08048434 <+19>:   leave // Primeiro passo do epílogo – desaloca memória e recupera o frame pointer do stack frame anterior.
  0x08048435 <+20>:   ret
End of assembler dump.
(gdb) i r $esp $ebp
esp            0xbffffb88      0xbffffb88
ebp            0xbffffb98      0xbffffb98
(gdb) ni
0x08048435 in quadrado ()
(gdb) disassemble quadrado
Dump of assembler code for function quadrado:
  0x08048421 <+0>:    push   %ebp
  0x08048422 <+1>:    mov    %esp,%ebp
  0x08048424 <+3>:    sub    $0x10,%esp
  0x08048427 <+6>:    mov    0x8(%ebp),%eax
  0x0804842a <+9>:    imul   0x8(%ebp),%eax
  0x0804842e <+13>:   mov    %eax,-0x4(%ebp)
  0x08048431 <+16>:   mov    -0x4(%ebp),%eax
  0x08048434 <+19>:   leave
=> 0x08048435 <+20>:   ret // Pega o valor que está no topo da stack (endereço de retorno), coloca em EIP e continua a execução.
End of assembler dump.
(gdb) i r $esp $ebp
esp            0xbffffb9c      0xbffffb9c // ESP anterior menos 4 (do pop %ebp).
ebp            0xbffffbc8      0xbffffbc8 // Frame pointer do stack frame anterior (função soma()).
(gdb) x/x $esp
0xbffffb9c:    0x0804841c // Endereço para onde retornar. A instrução ret retira este endereço do topo da stack, coloca no registrador EIP e desta forma a execução do programa continua.
(gdb) i r $eip
eip            0x8048435       0x8048435 <quadrado+20> // EIP aponta para a próxima instrução a ser executada, que no caso é a instrução ret da função quadrado().
(gdb) ni
0x0804841c in soma ()
(gdb) disassemble soma
Dump of assembler code for function soma:
  0x08048400 <+0>:    push   %ebp
  0x08048401 <+1>:    mov    %esp,%ebp
  0x08048403 <+3>:    sub    $0x28,%esp
```

```
  0x08048406 <+6>:    mov   0xc(%ebp),%eax
  0x08048409 <+9>:    mov   0x8(%ebp),%edx
  0x0804840c <+12>:   add   %edx,%eax
  0x0804840e <+14>:   mov   %eax,-0xc(%ebp)
  0x08048411 <+17>:   mov   -0xc(%ebp),%eax
  0x08048414 <+20>:   mov   %eax,(%esp)
  0x08048417 <+23>:   call  0x8048421 <quadrado>
=> 0x0804841c <+28>:   mov   -0xc(%ebp),%eax // Copia resultado da soma (conteúdo da variável res) em EAX, que é como a função retorna valor ao chamador.
  0x0804841f <+31>:   leave
  0x08048420 <+32>:   ret
End of assembler dump.
(gdb) i r $esp $eip
esp            0xbffffba0      0xbffffba0 // ESP decrementado de 4 depois que o conteúdo do seu topo foi colocado no EIP.
eip            0x804841c       0x804841c <soma+28> // EIP aponta para a instrução dentro da função soma(), que é a subsequente a da chamada para quadrado(). Logo o fluxo normal de execução do programa é retomado.
(gdb)
```

Listagem 5 – Execução passo a passo do epílogo da função quadrado() do programa stack_frame

NA IA-32, é o registrador EIP que faz o papel do contador de programa (*program counter*)[12], ou seja, ele aponta para a próxima instrução a ser executada. O endereço que a instrução **call** empilha na *stack* é o conteúdo de EIP, onde este endereço salvo é utilizado posteriormente no epílogo da função de forma a garantir que a execução do programa continue após o termino da função. O epílogo possui duas instruções, que são:

"leave
ret"

A instrução **leave** copia o conteúdo de EBP para o ESP, fazendo com que o topo da *stack* aponte para o *frame pointer* salvo do *stack frame* anterior. Em seguida ela desempilha o *frame pointer* salvo e o armazena em EBP, ou seja, atualiza o *frame pointer* atual para apontar para o *stack frame* anterior. A Tabela 1 mostra uma decomposição do que a instrução **leave** executa.

A instrução **ret** desempilha o conteúdo do topo da pilha, neste caso o endereço de retorno salvo quando ocorreu à execução da instrução **call**, colocando seu valor no registrador EIP, e em seguida a execução do programa continua. Cabe notar que se um usuário conseguir controlar o conteúdo do endereço de retorno em qualquer *stack frame,* quando for executado o retorno desta função ele controlará o fluxo de execução do programa e poderá executar códigos arbitrários dentro do contexto da aplicação explorada. Geralmente este é um dos objetivos das explorações de corrupção de memória.

O registrador EIP não pode ser referenciado diretamente, logo não é possível mostrar uma decomposição da instrução **ret** com instruções reais válidas. Mas como forma de facilitar o entendimento desta instrução, a Tabela 1 usa abstrações de lógica para mostrar o que ela executa.

[12]- *Program Counter é um termo genérico utilizado na literatura de teoria de arquitetura de computadores para referenciar o registrador responsável por apontar para o endereço da próxima instrução a ser executada, que na arquitetura aqui descrita (IA-32) trata-se do registrador EIP.*

Instrução	Equivale a:
leave	mov %ebp, %esp pop %ebp
ret	EIP <= (%esp); ESP <= ESP + 4; Prossegue a execução

Tabela 1 – Equivalência de Instruções

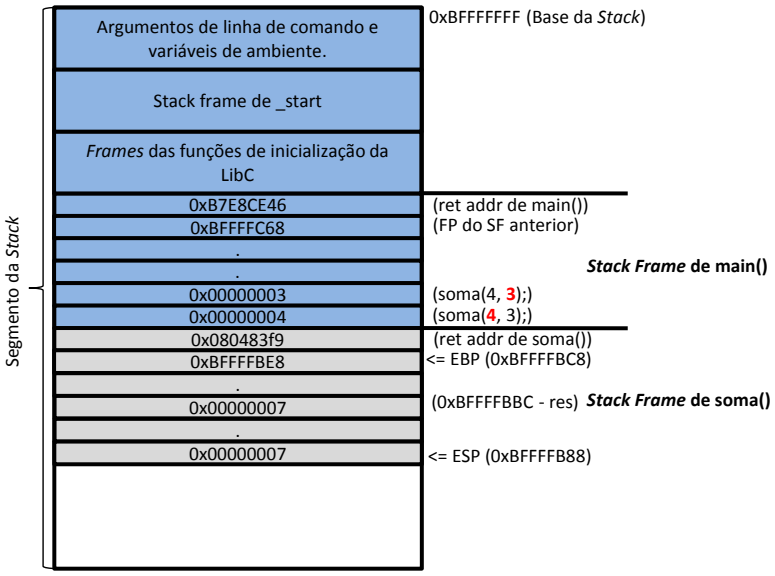


Figura 9 – Organização dos stack frames do programa stack_frame, depois da execução do epílogo da função quadrado()

A Figura 9 mostra como ficou o empilhamento dos stack frames após o retorno da função quadrado().

A Listagem 6 mostra a execução passo a passo do epílogo das funções soma() e main().

```
(gdb) disassemble soma
Dump of assembler code for function soma:
  0x08048400 <+0>:    push   %ebp
  0x08048401 <+1>:    mov    %esp,%ebp
  0x08048403 <+3>:    sub    $0x28,%esp
  0x08048406 <+6>:    mov    0xc(%ebp),%eax
  0x08048409 <+9>:    mov    0x8(%ebp),%edx
  0x0804840c <+12>:   add    %edx,%eax
  0x0804840e <+14>:   mov    %eax,-0xc(%ebp)
  0x08048411 <+17>:   mov    -0xc(%ebp),%eax
  0x08048414 <+20>:   mov    %eax,(%esp)
  0x08048417 <+23>:   call   0x8048421 <quadrado>
=> 0x0804841c <+28>:   mov    -0xc(%ebp),%eax
  0x0804841f <+31>:   leave
  0x08048420 <+32>:   ret
End of assembler dump.
(gdb) x/x $ebp-0xc
0xbffffbbc:    0x00000007 // Conteúdo da soma – variável local ret.
(gdb) i r $eax
eax            0x31    49 // Valor retornado da função quadrado().
(gdb) ni
0x0804841f in soma ()
(gdb) disassemble soma
Dump of assembler code for function soma:
  0x08048400 <+0>:    push   %ebp
  0x08048401 <+1>:    mov    %esp,%ebp
  0x08048403 <+3>:    sub    $0x28,%esp
```

```
  0x08048406 <+6>:    mov   0xc(%ebp),%eax
  0x08048409 <+9>:    mov   0x8(%ebp),%edx
  0x0804840c <+12>:   add   %edx,%eax
  0x0804840e <+14>:   mov   %eax,-0xc(%ebp)
  0x08048411 <+17>:   mov   -0xc(%ebp),%eax
  0x08048414 <+20>:   mov   %eax,(%esp)
  0x08048417 <+23>:   call  0x8048421 <quadrado>
  0x0804841c <+28>:   mov   -0xc(%ebp),%eax
=> 0x0804841f <+31>:   leave // Inicio do epílogo.
  0x08048420 <+32>:   ret
End of assembler dump.
(gdb) i r $eax
eax            0x7     7 // Retornando o resultado da soma.
(gdb) i r $esp $ebp
esp            0xbffffba0      0xbffffba0
ebp            0xbffffbc8      0xbffffbc8
(gdb) ni
0x08048420 in soma ()
(gdb) disassemble soma
Dump of assembler code for function soma:
  0x08048400 <+0>:    push  %ebp
  0x08048401 <+1>:    mov   %esp,%ebp
  0x08048403 <+3>:    sub   $0x28,%esp
  0x08048406 <+6>:    mov   0xc(%ebp),%eax
  0x08048409 <+9>:    mov   0x8(%ebp),%edx
  0x0804840c <+12>:   add   %edx,%eax
  0x0804840e <+14>:   mov   %eax,-0xc(%ebp)
  0x08048411 <+17>:   mov   -0xc(%ebp),%eax
  0x08048414 <+20>:   mov   %eax,(%esp)
  0x08048417 <+23>:   call  0x8048421 <quadrado>
  0x0804841c <+28>:   mov   -0xc(%ebp),%eax
  0x0804841f <+31>:   leave
=> 0x08048420 <+32>:   ret // Pega o valor que está no topo da stack, coloca em EIP e a execução continua.
End of assembler dump.
(gdb) i r $esp $ebp
esp            0xbffffbcc      0xbffffbcc // ESP anterior menos 4 (do pop %ebp).
ebp            0xbffffbe8      0xbffffbe8 // Frame pointer do stack frame anterior (função main()).
(gdb) x/x $esp
0xbffffbcc:    0x080483f9 // Endereço para onde retornar. A instrução ret retira este endereço do topo da stack, coloca
no registrador EIP e a execução continua.
(gdb) ni
0x080483f9 in main ()
(gdb) disassemble main
Dump of assembler code for function main:
  0x080483dc <+0>:    push  %ebp
  0x080483dd <+1>:    mov   %esp,%ebp
  0x080483df <+3>:    and   $0xfffffff0,%esp
  0x080483e2 <+6>:    sub   $0x10,%esp
  0x080483e5 <+9>:    movl  $0x3,0x4(%esp)
  0x080483ed <+17>:   movl  $0x4,(%esp)
  0x080483f4 <+24>:   call  0x8048400 <soma>
=> 0x080483f9 <+29>:   mov   $0x0,%eax // Valor retornado por main().
  0x080483fe <+34>:   leave
  0x080483ff <+35>:   ret
```

```
End of assembler dump.
(gdb) i r $esp $eip $eax
esp            0xbffffbd0      0xbffffbd0 // ESP decrementado de 4 depois que o conteúdo do seu topo foi colocado no EIP.
eip            0x80483f9       0x80483f9 <main+29> // EIP apontando para instrução dentro da função main(), seguinte a chamada para soma(), retomando o fluxo normal de execução do programa.
eax            0x7     7 // Valor retornado pela função soma().
(gdb) ni
0x080483fe in main ()
(gdb) disassemble main
Dump of assembler code for function main:
   0x080483dc <+0>:     push   %ebp
   0x080483dd <+1>:     mov    %esp,%ebp
   0x080483df <+3>:     and    $0xfffffff0,%esp
   0x080483e2 <+6>:     sub    $0x10,%esp
   0x080483e5 <+9>:     movl   $0x3,0x4(%esp)
   0x080483ed <+17>:    movl   $0x4,(%esp)
   0x080483f4 <+24>:    call   0x8048400 <soma>
   0x080483f9 <+29>:    mov    $0x0,%eax
=> 0x080483fe <+34>:    leave // Início do epílogo de main().
   0x080483ff <+35>:    ret
End of assembler dump.
(gdb) i r $eax
eax            0x0     0 // Valor retornado por main().
(gdb) i r $esp $ebp
esp            0xbffffbd0      0xbffffbd0
ebp            0xbffffbe8      0xbffffbe8
(gdb) ni
0x080483ff in main ()
(gdb) disassemble main
Dump of assembler code for function main:
   0x080483dc <+0>:     push   %ebp
   0x080483dd <+1>:     mov    %esp,%ebp
   0x080483df <+3>:     and    $0xfffffff0,%esp
   0x080483e2 <+6>:     sub    $0x10,%esp
   0x080483e5 <+9>:     movl   $0x3,0x4(%esp)
   0x080483ed <+17>:    movl   $0x4,(%esp)
   0x080483f4 <+24>:    call   0x8048400 <soma>
   0x080483f9 <+29>:    mov    $0x0,%eax
   0x080483fe <+34>:    leave
=> 0x080483ff <+35>:    ret
End of assembler dump.
(gdb) i r $esp $ebp
esp            0xbffffbec      0xbffffbec // ESP anterior menos 4 (do pop %ebp).
ebp            0xbffffc68      0xbffffc68 // Frame pointer do stack frame anterior (procedimento de inicialização da LibC).
(gdb) x/x $esp
0xbffffbec:    0xb7e8be46 // Endereço para onde retornar. A instrução ret retira este endereço do topo da stack, coloca no registrador EIP e a execução continua.
(gdb) ni
0xb7e8be46 in __libc_start_main () from /lib/i386-linux-gnu/i686/cmov/libc.so.6 // Neste ponto o programa que escrevemos já terminou sua execução, e agora passou a bola para o procedimento de inicialização da LibC, onde a função main() foi chamada.
(gdb) i r $eip
eip            0xb7e8be46      0xb7e8be46 <__libc_start_main+230> // Confirmado que o fluxo de execução está realmente na função de inicialização da LibC.
(gdb)
```

Listagem 6 – Execução passo a passo dos epílogos das funções soma() e main() no programa stack_frame

Após a execução do epílogo de main, o fluxo de execução é passado para a rotina de inicialização da LibC, que executa mais algumas verificações e encerra o programa. O que esta rotina de inicialização executa é assunto para um próximo artigo. :)

0x1 – Conclusão

Este artigo, composto por duas partes, mostrou um pouco da complexidade que envolve a execução de um programa, e a troca de contexto entre os procedimentos/funções que o compõe. O conhecimento aqui apresentado é útil em diversas áreas da computação, como análise de *malware*, análise de *crash*, escrita de *cracks*, exploração de vulnerabilidades de corrupção de memória e outras. Por ser apenas um conhecimento pontual dentro dos diversos necessários para se explorar um *software*, ele também mostra o quão complexo é esta área de quebra de sistemas.

O segredo para se explorar *software* é conhecer como as coisas realmente funcionam, pois não tem como você encontrar erros em um sistema se você nem sabe como ele deveria funcionar, e o entendimento de como é manipulado os *stack frames* é essencial para quem quer seguir no ramo de computação ofensiva.

Os autores e a revista se preocupam com a corretude das informações aqui apresentadas. Se você encontrou algum erro ou gostaria de agregar alguma informação às apresentadas aqui, por favor, deixe-nos saber. Sugestões de melhoria ou de assuntos a abordar são muito bem vindas.

0x2 – Perguntas Mais Frequentes

Questão 1: Vocês disseram que a ABI define que os argumentos devem ser passados na *stack* em ordem inversa. Existe algum motivo para isto?

Resposta 1: Sim. Isto é feito para auxiliar funções que possuem um número variável de argumentos, como printf() por exemplo. Geralmente nestas funções o seu primeiro argumento armazena um indicador de quantos argumentos a função está recebendo. Como o primeiro argumento está sempre no topo da *stack*, o mesmo sempre pode ser acessado de forma precisa para se conhecer a quantidade de argumentos da chamada corrente à referida função.

Q 2: Na execução passo a passo da função main() no programa stack_frame, porque você não criou um *break point* direto no símbolo *main,* mas ao invés disso preferiu desmontar o código para pegar o endereço inicial?

R 2: A ideia aqui era executar passo a passo todas as instruções desta função, e caso fosse criado um *"break main"*[13] o gdb iria parar na terceira instrução, logo após a execução do prólogo. A princípio isto pode parecer errado, mas faz um pouco de sentido quando você começa a entender que o prólogo só ajusta o *frame pointer* para apontar para o *stack frame* criado.

Q 3: Na execução passo a passo da função main(), como vocês sabiam que a instrução de retorno estava logo abaixo da instrução __libc_start_main+227?

R 3: Não sabíamos. Foi verificado antes de usar o *disassembler* no local exato. Isto foi feito como forma de otimizar a visualização de acordo com a necessidade do artigo.

---

[13]- *Durante o processo de revisão do artigo, o amigo (e revisor) Gabriel Negreira forneceu uma dica que os autores desconheciam até então. Caso se use 'break *main', o break-point é criado na primeira instrução do símbolo main.*

Q 4: Vi uma palestra de um especialista sobre *exploits* para ambiente Linux, onde o palestrante informa que "é óbvio que a primeira função executada em um programa feito em C é a main()" (http://www.youtube.com/watch?list=PL3724990A515A780F&feature=player_detailpage&v=bojlT4iQgSU#t=357). Já neste artigo vocês informam que outras coisas são executadas antes de se executar a main(). Afinal, quem está certo? Vocês conseguem provar o que dizem?

R 4: É possível que este especialista não seja lá tão especialista como diz. Se mesmo com a análise do endereço de retorno de main() você não se convenceu, ainda é possível fazer um teste simples como forma de se comprovar isto. Segue exemplo usando nosso código:

```
root@research:~/H2HC_Magazine/Fundamentos/Stack_Frame# gcc -o stack_frame stack_frame.c // Compilando
nosso fonte.
root@research:~/H2HC_Magazine/Fundamentos/Stack_Frame# readelf -h stack_frame | grep Entry // Verificando
qual o endereço do entry-point do binário gerado.
  Entry point address:               0x80482f0 // Este é o endereço do entry-point.
root@research:~/H2HC_Magazine/Fundamentos/Stack_Frame# objdump -d stack_frame | grep 80482f0 -A 12
// Verificando o que tem no endereço do entry-point.
080482f0 <_start>: // Aqui fica claro que o entry-point não aponta para o símbolo main, mas sim para o símbolo
_start.
 80482f0:      31 ed              xor    %ebp,%ebp
 80482f2:      5e                 pop    %esi
 80482f3:      89 e1              mov    %esp,%ecx
 80482f5:      83 e4 f0           and    $0xfffffff0,%esp
 80482f8:      50                 push   %eax
 80482f9:      54                 push   %esp
 80482fa:      52                 push   %edx
 80482fb:      68 40 84 04 08     push   $0x8048440
 8048300:      68 50 84 04 08     push   $0x8048450
 8048305:      51                 push   %ecx
 8048306:      56                 push   %esi
 8048307:      68 dc 83 04 08     push   $0x80483dc // Endereço de main (vide abaixo).
 804830c:      e8 cf ff ff ff     call   80482e0 <__libc_start_main@plt> // Isto mostra que mesmo antes da
execução do procedimento de inicialização da LibC, é executado outros códigos.
root@research:~/H2HC_Magazine/Fundamentos/Stack_Frame# objdump -d stack_frame | grep "<main>" -A 8
// Localizando endereço do símbolo de main.
080483dc <main>: // Aqui está o endereço de main. Isto mostra claramente que é um endereço diferente do entry-
point apontado no cabeçalho do binário ELF.
 80483dc:      55                 push   %ebp
 80483dd:      89 e5              mov    %esp,%ebp
 80483df:      83 e4 f0           and    $0xfffffff0,%esp
 80483e2:      83 ec 10           sub    $0x10,%esp
 80483e5:      c7 44 24 04 03 00 00    movl   $0x3,0x4(%esp)
 80483ec:      00
 80483ed:      c7 04 24 04 00 00 00    movl   $0x4,(%esp)
 80483f4:      e8 07 00 00 00     call   8048400 <soma>
root@research:~/H2HC_Magazine/Fundamentos/Stack_Frame#
```

O exemplo acima mostra claramente que antes de main é executado uma série de outras instruções, e ainda é possível confirmar que o endereço de main() é passado como primeiro argumento para o procedimento de inicialização da LibC.

O ponto principal não é quem está certo, mas sim que devemos fazer nós mesmos nossos testes e tirar nossas conclusões, ao invés de ficar acreditando em "especialistas".

Q 5: Este mesmo especialista ainda informa nesta palestra que o EBP armazena o endereço da base da memória, e que o ESP armazena o endereço da localização da *stack* (http://www.youtube.com/watch?list=PL3724990A515A780F&feature=player_detailpage&v=bojlT4iQgSU#t=301). Provavelmente ele deve ter se enganado novamente. Confere?

R 5: Sim. Ao longo do ciclo de execução de um processo no Linux, o EBP em momento algum deve apontar para a base da memória[14]. E quanto a localização da *stack*, sabemos que em ambientes não randomizados ela sempre começa no endereço 0xBFFFFFFF e o ESP aponta para o seu topo, e não para sua base.

Q 6: O mesmo especialista nesta palestra ainda diz que "é obvio que as funções são carregadas dentro da *stac*k" (http://www.youtube.com/watch?list=PL3724990A515A780F&feature=player_detailpage&v=bojlT4iQgSU#t=469). Neste artigo, apesar de se falar de *stack* e *stack frames*, não se diz nada de onde as funções são carregadas. A informação deste especialista procede?

R 6: Não. Quando um programa é criado em linguagem de máquina, ele é dividido em dados e código (e outras partes também, mas que não tem importância para o contexto desta pergunta). O que é código vai para um segmento conhecido como text, e que possui permissão de execução. É aqui que as funções são carregadas.
Nas arquiteturas dos sistemas operacionais modernos nem é mais possível se executar código no segmento da *stack,* ou seja, mesmo que fosse verdade o que o palestrante disse o código não poderia ser executado.

Devemos tomar muito cuidado com o que estes pseudo-especialistas dizem por aí, ainda mais nesta área de computação. O melhor a se fazer é estudar e testar o que o material está falando. É como diz o nosso amigo barba (http://techtechpodcast.com/2013/08/hackerismo.html - 1:44:16hs), quem é sênior é sênior.

Q 7: Sobre o formato do *stack frame* nesta arquitetura. Diferente do mostrado aqui, existem diversos livros que dizem que os argumentos estão contidos no *stack frame* corrente (Hacking – The Art of Exploitation 2nd Edition, pág 73; A Guide To Kernel Exploitation, pág 54; Practical Malware Analysis, pág 79). Vocês não teriam se enganado quanto ao *layout*?

R 7: Não. Pode conferir na ABI do Linux para a referida arquitetura e no manual da Intel (SYSTEM V APPLICATION BINARY INTERFACE - Intel386 Architecture Processor Supplement - Fourth Edition, pág 3-10; Manual da Intel - volume 1, pág 6-5).

Q 8: Isso quer dizer que estes livros são ruins?

R 8: Definitivamente não. Isto apenas mostra o quão difícil é escrever informação de qualidade. Dos livros referenciados, nós mesmos recomendamos dois deles (Hacking – The Art of Exploitation 2nd Edition e o A Guide To Kernel Exploitation). São excelentes livros, apesar destes pequenos erros.

---

[14]- *Qualquer código que possa ser encontrado no programa que limpe o EBP não o faz com o intuito de utilizá-lo como ponteiro, mas sim de remover possível lixo presente em tal registrador para iniciar seu uso apontando para posições dentro da stack.*

Q 9: Ainda sobre o *layout* do *stack frame*, ele sempre terá o formato aqui descrito?

R 9: Não. O formato aqui descrito é o mais comum e utilizado pelos sistemas atuais, mas isto pode variar entre compiladores diferentes. O que muda é que alguns compiladores criam funções sem o uso do *frame pointer*, nestes casos as variáveis locais e os parâmetros são referenciados de outras formas. Determinadas opções de compilação, como as de otimização, também podem mudar o *layout* do *stack frame*. De toda forma, os argumentos para o procedimento chamado sempre estará contido no *stack frame* do chamador.

YGOR DA ROCHA PARREIRA

Ygor da Rocha Parreira faz pesquisa com computação ofensiva, trabalha como consultor de segurança de aplicações na Trustwave e é um cara que prefere colocar os bytes à frente dos títulos.

FILIPE BALESTRA

Filipe Balestra é especialista em segurança da informação, onde está envolvido desde 1997. Executou projetos de segurança em diversas empresas, é organizador do H2HC além de editor desta revista. Publicou falhas de segurança em importantes *software*, bem como artigos em locais como Hakin9 e Phrack Magazine.

Sessão Renegada

GODZILLA

Direção: Gareth Edwards
Elenco: Aaron Taylor-Johnson, Akira Takarada, Al Sapienza, Brian Markinson, Bryan Cranston, Carson Bolde, Chris West, Christian Tessier, CJ Adams, Dan Zachary, David Strathairn, Elizabeth Olsen, Jake Cunanan, Jeric Ross, Juliette Binoche, Ken Watanabe, Ken Yamamura, Patrick Sabongui, Peter Dwerryhouse, Primo Allon, Raj K. Bose, Richard T. Jones, Sally Hawkins, Victor Rasuk, Warren Takeuchi, Yuki Morita.
Roteiro: David Callaham, David S. Goyer, Max Borenstein
Produção: Brian Rogers, Dan Lin, Jon Jashni, Roy Lee, Thomas Tull
Edição: Bob Ducsay
Trilha Sonora: Alexandre Desplat
Fotografia: Seamus McGarvey
Gênero: Ação
País: EUA
Duração: 123 min.
Ano: 2014
Estúdio: Legendary Pictures / Lin Pictures / Toho Company / Vertigo Entertainment / Warner Bros. Pictures
Classificação: 12 anos

SINOPSE

Joe Brody (Bryan Cranston) criou o filho sozinho após a morte da esposa (Juliette Binoche) em um acidente na usina nuclear em que ambos trabalhavam, no Japão. Ele nunca aceitou a catástrofe e quinze anos depois continua remoendo o acontecido, tentando encontrar alguma explicação. Ford Brody (Aaron Taylor-Johnson), agora adulto, é soldado do exército americano e precisa lutar desesperadamente para salvar a população mundial – e em especial sua família – do gigantesco, inabalável e incrivelmente assustador monstro Godzilla.

CRÍTICA RENEGADA

O que falar sobre um dos filmes mais esperados do ano?
Infelizmente nem tudo acabou sendo flores para Godzilla. O problema não seriam os monstros, mas as pessoas desse filme. Por mais que o filme pegue bem a base japonesa das histórias tradicionais do monstro, deram algumas explicações plausíveis em alguns eventos históricos reais justificando algumas coisas com os monstros, e os efeitos são muito bons, o drama que segue depois de um acontecimento com um personagem que se mostrava interessante desde o começo do filme, a parte dos humanos fica bem fraquinha.

Aaron Taylor-Johnson (o Kick-ass) não tem o peso necessário para segurar todo o drama mais sério do filme. Você fica impaciente pelo jeito que ele vai tratando e as coisas começam rolar por causa dele, acaba sendo bem aquele "clichêzão" foda que sempre os americanos solucionariam toda essa confusão, ponto que eu acabei não gostando no filme também. Você não vê a preocupação de outras nações com o surgimento de bestas colossais, uma coisa que seria bem bacana de ser explorada, afinal, são monstros gigantes que não ligam para os seres humanos (ou não??).

Porém, Godzilla está impecável na tela, assim como os outros seres que ele busca durante o filme, a sutileza da revelação de todos esses e as surpresas nas brigas. Godzilla esta no tamanho ideal para os cinemas! Uma coisa que eu havia reclamado também, mas parece que só veremos tal cena nos Blu-rays por aqui seria uma cena para puxar uma continuação mais do que obrigatória desse filme. PS: Ignorem o filme 3D e o antigo Godzilla Americano.

Por: Alessandro "Bob" Bernard

NO LIMITE DO AMANHÃ

Direção: Doug Liman
Elenco: Bill Paxton, Brendan Gleeson, Charlotte Riley, Dragomir Mrsic, Emily Blunt, Franz Drameh, Jeremy Piven, Jonas Armstrong, Kick Gurry, Madeleine Mantock, Tom Cruise, Tony Way
Roteiro: Alex Kurtzman, Christopher McQuarrie, Joby Harold, Roberto Orci
Produção: Erwin Stoff, Gregory Jacobs, Jason Hoffs, Joby Harold
Edição: James Herbert
Trilha Sonora: Christophe Beck
Fotografia: Dion Beebe
Gênero: Ação
País: EUA
Duração: 113 min.
Ano: 2014
Estúdio: Warner Bros. Pictures
Classificação: 14 anos

SINOPSE

Quando a Terra é tomada por alienígenas, Bill Cage (Tom Cruise), relações públicas das Forças Armadas dos Estados Unidos, é obrigado a ir para a linha de frente no dia do confronto final. Inexplicavelmente ele acaba preso no tempo, condenado a reviver esta data repetidamente. A cada morte e renascimento, Cage avança e, antecipando os acontecimentos, tem a chance de mudar o curso da batalha com o apoio da guerreira Rita Vrataski (Emily Blunt).

CRÍTICA RENEGADA

Quando comecei a ver o trailer desse filme, não sabia o que esperar, mesmo ja sacando que se tratava em viagem no tempo em looping. E QUE GRATA SURPRESA!!

Não esperava que o filme acabasse sendo tão dinâmico, divertido e com muitos tiros! Como alguns já disseram por ai, um video-game com checkpoints. Alguns ainda ficam assim pelo fato de não terem gostado muito dos últimos filmes do Tom Cruise, mas aqui a diversão é garantida!

Ele corre? claro! E quando chega nessa parte, você já estará pensando (Que FO@#@! quero um Exoesqueleto de combate logo). A motivação de saber que adquiriu o dom e aproveitar para usa-lo nessa guerra, as sacadas das tentativas, entre outras coisas enriquecem o drama, até mesmo quando ele conhece a personagem FODONA da Emily Brunt... O Jeito que ela é retratada e tamanho respeito acabam deixando você Boquiaberto. Os invasores (Uma pegada estilo Tropas Estrelares) são aterrorizantes pelo fato da sua movimentação e estratégias de combates, o que acaba deixando inesperada qualquer reação das tropas aliadas. O Design deles estão muito bons. Bem, finalizando, fiquei sabendo que o filme é adaptação do romance "All You Need is Kill", de Hiroshi Sakurazaka e Brad Pitt foi cotado para o papel principal. Uma coisa que acabei pensando depois foi que Elysium queria ser muito esse filme.

Por: Alessandro "Bob" Bernard

TRANSCENDENCE – A REVOLUÇÃO

Direção: Wally Pfister
Elenco: Johnny Depp, Rebecca Hall, Morgan Freeman, Paul Bettany, Cillian Murphy, Kate Mara, Cole Hauser, Clifton Collins Jr.
Roteiro: Jack Paglen
Produção: Andrew A. Kosove, Annie Marter, Broderick Johnson, Kate Cohen, Marisa Polvino
Gênero: Ficção Científica
País: EUA
Duração: 119 min.
Ano: 2014
Estúdio: Alcon Entertainment / Straight Up Films
Classificação: 12 anos

SINOPSE

O dr. Will Caster (Johnny Depp) é o mais famoso pesquisador sobre inteligência artificial da atualidade. No momento ele está trabalhando na construção de uma máquina consciente que conjuga informações sobre todo tipo de conteúdo com a grande variedade de emoções humanas. O fato de se envolver sempre em projetos controversos fez com que Caster ganhasse notoriedade, mas ao mesmo tempo o tornou o inimigo número 1 dos extermistas que são contra o avanço da tecnologia – e por isso mesmo tentam detê-lo a todo custo. Só que um dia, após uma tentativa de assassinato, Caster convence sua esposa Evelyn (Rebecca Hall) e seu melhor amigo Max Waters (Paul Bettany) a testar seu novo invento nele mesmo. Só que a grande questão não é se eles podem fazer isto, mas se eles devem dar este passo.

CRÍTICA RENEGADA

Quando fui ver esse filme, esperava ver um Johnny Depp fora dos papéis tradicionais. E Simplificando bem porque o filme acaba tendo aquele ar de importância, mas se perde em um Roteiro bem fraco e vira um filme qualquer. Não adianta os efeitos e a ideia da Inteligência Artificial, o carisma e diversão que o filme poderia ter tomado se perde pela tentativa de ser um filme cabeça/ dramático.

Sem contar que o filme acaba sendo muito previsível, sem nenhuma surpresa. Nem Morgan Freeman salva, passa totalmente batido. Rebecca Hall acaba fazendo quase o mesmo papel que tinha feito em Homem de Ferro 3, da cientista que luta para ficar com o seu amado e a cagada do amor fodo quase com a humanidade inteira. Cenas que poderiam dar um ar mais sério e dramático se perdem pela resolução do problema e a facilidade de tudo. É, eu tentei mesmo gostar do filme, mas a sensação depois de você ver um filme desse é de um vazio estranho pela oportunidade de ser um filme mais direto e menos drama. Deixa sair pra alugar.

Por: Alessandro "Bob" Bernard

COMO TREINAR O SEU DRAGÃO 2

Direção: Dean DeBlois
Elenco: Gerard Butler, Jonah Hill, Christopher Mintz-Plasse, Kristen Wiig, Jay Baruchel, Kit Harington, America Ferrera, T. J. Miller, Craig Ferguson
Roteiro: Baseado na obra de Cressida Cowell, Dean DeBlois
Produção: Bonnie Arnold
Trilha Sonora: John Powell
Gênero: Animação
País: EUA
Duração: 102 min.
Ano: 2014
Estúdio: DreamWorks Animation / Mad Hatter Entertainment / Vertigo Entertainment
Classificação: Livre

SINOPSE

Cinco anos após convencer os habitantes de seu vilarejo que os dragões não devem ser combatidos, Soluço (voz de Jay Baruchel) convive com seu dragão Fúria da Noite, e estes animais integraram pacificamente a rotina dos moradores da ilha de Berk. Entre viagens pelos céus e corridas de dragões, Soluço descobre uma caverna secreta, onde centenas de novos dragões vivem. O local é protegido por Valka (voz de Cate Blanchett), mãe de Soluço, que foi afastada do filho quando ele ainda era um bebê. Juntos, eles precisarão proteger o mundo que conhecem do perigoso Drago Bludvist (Djimon Hounson), que deseja controlar todos os dragões existentes.

CRÍTICA RENEGADA

Conversando com algumas pessoas após o filme, chegamos em uma conclusão: Que filme belo e grandioso da DreamWorks, muito mais trabalhado, tanto cenário e personagens, como tem uma ampliação do universo e mitologia do primeiro filme. A sinergia de Soluço e Banguela cativa pela viagem e determinação de mostrar ao público ate onde o mundo pode ir, tanto que vemos Solução fazendo anotações em um mapa incompleto, treinando manobras e buscando novas sofisticações para suas próprias dificuldades.

Quando chega o segundo ato do filme, quando é mostrado o grande "Alpha" dos Dragões, galera, que visual foda. O Alpha é gigantesco e la é mostrado a tamanha importância de um deles naquele vale. A aparição dele é de uma força imensa que não tem como você falar algum palavrão de intensidade quando o Dragão começa a se mexer... É de ficar maravilhado como ele casa com os dragões menores, com cores, casando tudo na tela. Os detalhes das armaduras dos personagens, cabelos, cenário, muito bom. Agora a história é realmente bem regular, a motivação de uma determinada personagem não convence muito, e tem até um momentinho Godzilla. Concluindo: Vale a pena sim, mas não espere tanto uma comediazinha sussa.

Por: Alessandro "Bob" Bernard

EA SPORTS - UFC

EA Sports UFC é um jogo de artes marciais mistas desenvolvido pela Eletronic Arts para Playstation 4 e Xbox One. O jogo é baseado no Ultimate Fighting Championship (UFC) e foi lançado em 17 de junho de 2014. Será o primeiro jogo da série desde que a THQ vendeu a licença a Eletronic Arts

JOGABILIDADE

Primeiro de tudo, esqueça hadoukens ou fatalitys. A EA mostra uma melhoria do que a THQ já fazia antes, dando uma refinada e deixando a visão de "simulador" para o UFC. Não rola de ficar sempre chutando ou apenas socos, aqui o jogador cansa e dependendo do Lutador seu peso influência nos Combates. Acaba sendo o que a gente vê na TV: Um jogo de Xadrez onde a melhor estratégia vence e qualquer vacilo pode acabar com sua felicidade em segundos. Para a proposta do game, a jogabilidade agrada e é bacana treinar pela variedade de golpes que você pode fazer.

VISUAL

O cuidado de simular uma transmissão é animal e totalmente em português! Videos, menu, tudo!! É como você realmente participasse de todo o show, desde as aberturas das lutas, como os seus resultados. O Menu do jogo te mostra tanta coisa que não consegue sair tão fluido como nos jogos do FIfa. As Texturas dos jogadores e movimentação estão muito boas para o 1º Jogo da Franquia no XOne. Você ve veias, pele sendo puxada, respirações ofegantes e demostrações de cansaço ou dor. Alguns lutadores ainda estão estranhos, mas passa de boa e a imersão é imediata.

MODO CARREIRA

Aqui foi a maior surpresa: O processo como você entra no UFC.E como você faz isso? Entrando e participando do TUF!! E o Bacana: Dana White vai mostrando sua satisfação (ou Frustração) por vídeos durante os treinos. Não só incentivos dele, mas de várias lendas novas (e velhas) falando sobre suas performances. Outra coisa que é bacana é q você vai ganhando pontos para evoluir seu personagem de acordo com suas lutas e ganhando Fãs para mais patrocínios. E Tem Várias situações como Lutadores mais consagrados irem treinar com você nos Intervalos de uma luta marcada para a outra. Muito bacana e eu ainda continuo jogando!

MULTIPLAYER

Pelo menos comigo ainda consigo jogar de boa em um multiplayer por causa da minha conexão que ajuda, e como um jogo como esse precisa de resposta mais imediata, pode acabar pecando um pouco. E funciona do mesmo jeito: Dando pontos para cada vez que você ganha e criando um Ranking para dividir com todos e curtir!

TREINAR, TREINAR E TREINAR

E é isso! O Bom de jogar o Modo carreira são os treinos... e uma vez dominando isso, a competitividade fica mais interessante e assim você consegue explorar outras formar de ganhar uma luta! E para jogar com os seus amigos (tanto online como na sua casa), treinar te deixa mais acostumado com o peso do seu lutador preferido e assim você desfruta de uma forma mais Fodástica do jogo...

AH, O BRUCE LEE...

Eu entendo a homenagem, mas sim, é bizarro jogar com ele. Vale a pena pela nostalgia, mas ele ta gigante e "mini-Craque"! Mas curta o personagem pq faz muito tempo que não veremos esse cara em qualquer outro jogo!

Por: Alessandro "Bob" Bernard

Estante Renegada

BATTLE ROYALE

SINOPSE

O livro Battle Royale pode ser definido com uma história Insanamente Divertida. Quarenta e dois estudantes japoneses que acreditam estar partindo para uma excursão de escola, são largados em uma ilha, equipados com armas, um kit de sobrevivência básico e somente uma instrução: apenas um de vocês sobreviverá.

CRÍTICA RENEGADA

Peguem sua metralhadora favorita, sua foice de estimação e seu kit de sobrevivência porque a partir de agora vou falar de uma das distopias mais malucas e sanguinárias já escritas: Battle Royale.

O Japão como você conhece já não existe mais. Ele faz parte da Grande República do Leste Asiático. Um país totalitário e repressor que para poder controlar o ímpeto dos jovens cria um programa em que os alunos de uma turma do ensino fundamental escolhida de forma aleatória, são levados para uma área isolada para que possam participar de um jogo. O jogo em si é simples, cada aluno recebe uma arma. Pode ser uma pistola, uma metralhadora ou uma espingarda. Mas também pode ser uma faca, uma tesoura e até mesmo um arco e flecha. Junto dessa arma é fornecido um kit de sobrevivência e a partir daí você está por conta própria. Apenas um aluno pode sobreviver.

Precisei ler apenas algumas páginas do livro para automaticamente ficar fã da história. A forma como o autor Koushi Takami narra os acontecimentos do jogo é sensacional. Aos poucos você vai conhecendo a vida de cada participante apenas para chegar no ponto em que você sinta algo quando ele toma uma flechada, ou um tiro na cabeça e até mesmo pule de um penhasco.

O Livro vai seguindo com os personagens morrendo capítulo a capítulo (na maioria das vezes mais que um de uma vez só) e em cada final aparece a frase “Restam 40 estudantes”, “Restam 31 estudantes” e assim vai até chegar no final com apenas um. Parece ser uma coisa simples mas que cria um ambiente ao ler o livro muito sombrio.
Em cada capítulo você vai conhecendo um pouco mais sobre a forma como o governo trata cada um dos alunos, suas famílias e até mesmo aqueles que comandam o jogo. Livro mais que recomendado a todos. Aliás, a todos com mais de 18 anos. Pois tem sexo, violência, sangue e tripas que deixaria Quentin Tarantino orgulhoso.

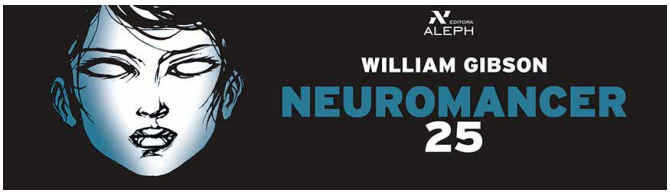
O mangá de Battle Royale foi publicado em japonês de 2000 a 2005 no Japão pela Akita Publishing e em inglês de 2003 a 2006 pela Tokyopop. No Brasil a publicação foi uma tremenda bagunça. A Conrad começou a publica-lo em 2006 mas interrompeu na edição 12. Continuou assim até 2011 quando retomou a produção e finalmente encerrou a saga. A história do mangá mantém basicamente a mesma do livro, mas se aprofunda mais na vida anterior dos personagens. Vale lembrar que ele foi todo escrito pelo mesmo autor do livro.

O Filme foi lançado em 2000 e manteve a brutalidade e violência presentes no livro. Em razão disso foi banido em vários países. Ainda teve uma continuação chamada de Battle Royale II: Requiem e até mesmo uma versão extendida denominada Battle Royale II: Revenge. Ambas as versões não tiveram o mesmo sucesso do primeiro filme. Uma das curiosidades desse filme foi que ele credenciou Chiaki Kuriyama a fazer o papel de Go Go Yubari em Kill Bill.

Termino essa resenha dizendo que esse livro foi a grande inspiração de Suzanne Collins para escrever Jogos Vorazes. Se você ler os dois livros, realmente percebe uma semelhança que chega a ser gritante.

Por: Marco "Mike" Dias

NEUROMANCER



SINOPSE

Um hacker renegado, uma samurai das ruas, um fantasma de computador, um terrorista psíquico e um rastafari orbital num thriller sexy, violento e intrigante. De Tóquio a Istambul, das estações espaciais ao não-espaço da realidade virtual, o tenso jogo final da humanidade contra as Inteligências Artificiais.

Neuromancer é o primeiro – e ainda hoje o mais famoso – livro de William Gibson. É considerado não só o romance que deu origem ao gênero cyberpunk, mas também o seu melhor representante.

CRÍTICA RENEGADA

Essa resenha dará um pulo no futuro para falar de uma das obras mais importantes e mais influentes das décadas 80 e 90: Neuromancer.

É um livro ousado que trouxe a tona temas como por exemplo: Inteligência Artificial, implantes corporais e uma distopia no estilo cyberpunk sensacional.

O livro foi publicado em 1984 e conta a historia de Case, um Cowboy (que são como os hackers são chamados) que em razão de uma invas mal sucedido perde o direito de se conectar a Matrix. E sim, o nome esta correto.

Case é envenenado com uma Toxina que danifica seu sistema neural e assim o impossibilita de se comunicar com a famosa rede e a partir de então vive a margem da sociedade desempregado e sem perspectiva de sucesso.

Quando uma mulher em roupas de couro chamada Molly tenta contrata-lo para realizar uma missão especial, Ele enxerga ali uma possibilidade de ser curado e ter sua antiga vida de volta.

No momento em que eu li Neuromancer pela primeira vez eu fiquei impressionado com tudo que o escritor sugeria como possibilidade de avanço tecnológico. Por mais que a historia por muitas vezes se arraste para contar algo é impressionante o tanto que ele influenciou e foi influenciado por obras marcantes literárias ou do cinema.

A influencia de Neuromancer em Matrix é absurda. Começando pela própria Matrix que no livro é como se fosse um grande servidor onde todos se conectam. Temos também o trio Neo, Trinity e Morpheus encontrado no livro em Case, Molly e Armitage. Molly é mulher misteriosa que usa roupas de couro agarrada e vem oferecer uma proposta a Case. Sim, ela é basicamente a Trinity (com a pequena diferença que ela tem implantes de olhos de inseto no lugar de seus olhos de verdade). Mas anos depois eu também fui descobrir que Molly foi a inspiração para criar Motoko, uma personagem do Mangá e Anime Ghost in the Shell.

Neuromancer é um clássico que obrigatoriamente deve ser lido. Ignore algumas partes chatas ou muito detalhistas do autor. Seu valor é inestimável para a cultura cyberpunk e muitas outras obras que vieram por sua causa.

Por: Marco "Mike" Dias

O JOGO INFINITO

SINOPSE

Michael é um gamer. E como a maioria dos jogadores, ele passa quase mais tempo no VirtNet do que no mundo real. O VirtNet oferece total imersão do corpo e da mente, e é viciante. Graças à tecnologia, qualquer pessoa com dinheiro suficiente pode experimentar mundos de fantasia, arriscar sua vida sem a chance de morte, ou apenas ficar com os virt-amigos. E quanto mais habilidades de hacker você tem, mais divertido. Por que se preocupar seguindo as regras quando a maioria delas são idiotas, afinal? Mas algumas regras foram feitas por uma razão. É muito perigoso brincar com algumas tecnologias. Há becos e esquinas no sistema que olhos humanos nunca viram e predadores que ele não pode nem mesmo imaginar – e há a possibilidade de que a linha entre jogo e realidade será borrada para sempre.

CRÍTICA RENEGADA

Quando vi esse livro pela primeira vez pensei: "Caraca!! Essa capa me lembra muito Inception (A Origem), parece ser um livro bom mesmo!"

Quando eu li a sinopse, foi amor a primeira vista! Games, Tecnologia, Realidade Virtual, Ação e Aventura em um só lugar?
Na minha cabeça já estava o cen'ario perfeito. Consegui um exemplar e mergulhei nesta outra realidade criada por James Dashner.

Como gamer, confesso que o universo criado pelo autor seria o que eu realmente gostaria de viver mas, claro, sem as partes assustadoras. Ah e por falar nisso, um pequeno mas importante detalhe: seja você um fissurado dos games ou não, vai encontrar no livro uma imersão surpreendentemente prazerosa.
Logo no primeiro capítulo somos apresentados a uma cena com Michael tentando salvar Tanya na Virtnet.

Ah sim, Virtnet é um mundo virtual, onde você pode fazer e ser qualquer coisa e para entrar neste ambiente maravilhoso, você precisa entrar no que eles chamam de Caixão (pense naquelas máquinas estilo Avatar).

Mas voltando à história, se você morre no ambiente do Virtnet, você sente toda a dor da morte mas no fim, sempre acorda novamente dentro do Caixão. Não há nenhum tipo de sequela ao seu corpo do mundo real. Mas Michael vê Tanya arrancando o Núcleo do seu personagem na Virtnet, através da manipulação do código do jogo. E aí você percebe que, se essa menina se jogar da ponte, a morte dela valeria no mundo real também. Mas o que que está acontecendo?!?!

E é nesse clima que o livro se desenrola. Em alguns momentos, eu tentei prever o que aconteceria no capítulo a seguir e tomei vários tapas na cara me dizendo que eu !

Aliás, caro leitor, lembre-se dessa frase: James Dashner não faz nada ao acaso!

Enfim, O Jogo Infinito é o começo de uma trilogia que eu recomendo muito, especialmente se você curte filmes como Inception, Avatar e Matrix ou até mesmo livros como Jogador nº 1.

Por: Mariana "Miho" Teruya

Horóscopo

Áries

Marte, Lua e Saturno em Escorpião podem trazer algum problema passageiro a venda de um 0day. Informe-se de novas regulações. Urano e Júpiter movimentam e beneficiam bug bounties.

Touro

A união de Marte, Lua e Saturno em Escorpião pode trazer desafios a conferências que organiza. Mantenha a calma e resolva o que for necessário, pois novos apoiadores começam a aparecer. Ótimas energias lhe ajudarão a se concentrar em suas sessões de Debugging, que lhe acalmam a alma e satisfazem suas ansiedades.

Gêmeos

Hoje, o dia de trabalho pode ser mais intenso e trazer alguns problemas e dificuldades passageiras para você resolver. Você será no entanto beneficiado com estas novas informações, aproveite o open-bar e resolva um problema de cada vez. Júpiter e Urano em ótimo aspecto movimentam agradavelmente sua vida social.

Câncer

É possível que você precise enfrentar uma dificuldade em seu código e que seu ego sofra por isso. No entanto, essa energia é passageira e você não deve super valorizá-la. Urano e Júpiter em ótimo aspecto movimentam sua vida profissional, com patches sendo desenvolvidos em tempo e os problemas eventualmente solucionados.

Leão

Com o sol favorecendo Leão, os malware deste signo tiveram suas prevalências expostas, mas continuarão influenciando seu astral. Excelente momento para repensar suas técnicas de prevenção. Urano e Júpiter em ótimo aspecto favorecem a exposição de ameaças.

Virgem

É possível que você sinta certa vontade de ficar mais na sua, em casa, ou mesmo somente dividindo sua vida com pessoas mais próximas. O momento é de maior introspecção e reflexão. Urano e Júpiter em ótimo aspecto ajudam no desenvolvimento de novos códigos. O maior evento e mais antigo evento da América Latina se aproxima e mudará este seu momento, e possivelmente sua visão do mundo para sempre.

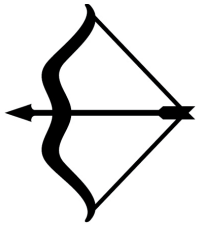
Libra

Diversos 0days serão reveleados com forte influência de Touro. O momento é ótimo para ganhos financeiros, apesar de todas as mudanças nas regulações e o aumento no controle das exportações. Júpiter indica que o momento é se concentrar em software de redes.

Escorpião

Marte, Saturno e Lua em seu signo podem tornar seu dia mais tenso, não se esqueça do open-bar e relaxe. Seus nervos podem estar à flor da pele, portanto, mantenha o autocontrole, aproveite e treine sua pontaria em uma Coruja. E não se preocupe, pois essa energia é passageira. Urano e Júpiter em ótimo aspecto movimentam positivamente o evento que mudará sua carreira.

Sagitário



Você pode estar mais fechado e voltado para emoções do passado. O dia é de limpeza e você deve deixar para trás o que precisa ir, esqueça os aprendizados errados que teve até aqui. Urano e Júpiter em ótimos aspecto entre si movimentam de maneira bastante positiva os aprendizados com estrangeiros. Novos projetos permeiam sua vida.

Capricórnio

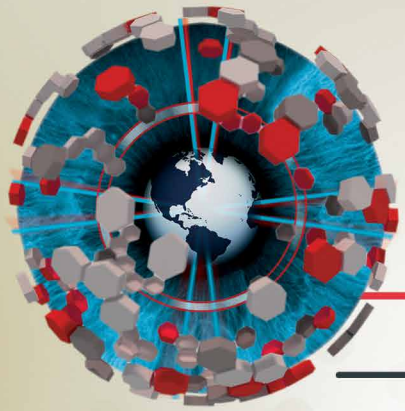
Hoje você estará mais voltado para si mesmo e avesso às questões que envolvem compromissos sociais. O momento é ótimo para organizar a gestão de um trabalho em equipe. Urano e Júpiter em ótimo aspecto entre si trazem o evento necessário para a descoberta de verdadeiras amizades e laços de pesquisa.

Aquário

Urano e Júpiter em ótimo aspecto entre si movimentam de maneira bastante positiva seus relacionamentos. O momento é ótimo para conhecer pessoas novas e começar uma pesquisa. Alguns pequenos problemas no trabalho podem chatear, mas não os leve tão a sério. Afinal, eles não participam do H2HC para entenderem realmente do que é segurança. Aqueles que criam as próprias arquiteturas e ensinam errados seus estudantes foram expostos mas persistirão em retornar.

Peixes

Marte, Lua e Saturno em Escorpião podem trazer algumas dificuldades de comunicação em um projeto de médio prazo que envolve o estrangeiros que conhecerá. Mantenha a calma, pois essa energia é passageira e você aprenderá a língua necessária para quebrar estas barreiras. Júpiter e Urano em ótimo aspecto melhoram os relacionamentos no trabalho, com colegas de empresa aproveitando o open-bar e relaxando com você.



H2HC

HACKERS TO HACKERS CONFERENCE

MAGAZINE