PROGRAMAS SEGUROS: VULNERABILIDADES COMUNS E CUIDADOS NO DESENVOLVIMENTO

Autor1	Autor2

RESUMO

O presente artigo trata de vulnerabilidades encontradas em programas supostamente seguros – principalmente programas SUID/SGID – e alguns cuidados a serem tomados no momento de desenvolver um programa seguro. Esta é uma área importante nestes dias em que é muito difundido o desenvolvimento de serviços de rede. A programação descuidada é a causa maior de erros nos programas e, conseqüentemente, de furos de segurança em sistemas computacionais, que podem vir a ser descobertos e explorados por atacantes conectados à Internet. Este trabalho dá uma breve visão sobre alguns estudos anteriores acerca deste tema. Nós tratamos sobre a linguagem C e o sistema UNIX, vastamente usados na Internet. Além disso, muitas ferramentas estão disponíveis para auxiliar os programadores a escreverem programas seguros, das quais nós apresentamos algumas.

ABSTRACT

This article addresses vulnerabilities found in supposedly secure programs – mainly SUID/SGID programs – and some guidelines to be considered during the development of secure programs. This is an important area in these days of widespread network services programming. Careless programming is the major cause of bugs and, consequently, of security holes in computer systems, which are discovered and exploited by attackers connected to the Internet. This work overviews some previous studies about this topic. We deal with the C language and the UNIX system, mostly used in the Internet. Also, many tools are available to help the programmers write safe programs, some of which we talk about.

1 INTRODUÇÃO

A programação segura de sistemas já vem sendo estudada há um bom tempo. O primeiro ataque da Internet – o *Worm*, em 1988 – explorava um erro de programação no *fingerd* (Spafford 1989). Depois deste ataque, o grupo de Berkeley verificou seu código e eliminou similares da função `gets()` dos servidores de rede – a explorada pelo *worm* – e outros fornecedores de UNIX também assim o fizeram (Garfinkel and Spafford 1996). Vários cientistas e programadores alertaram, na época, para o perigo do uso de outras funções semelhantes, tais como `strcpy()`.

A mais simples definição para o que seja um programa seguro é “um programa que é capaz de executar sua tarefa suportando quaisquer tentativas de subvertê-lo.” Além dos programas diretamente envolvidos com as políticas de segurança de um sistema – por exemplo o programa *login* do UNIX – também os programas que podem afetar essas políticas devem ser seguros (Al-Herbish 1999). A programação segura envolve programas especiais que atuam em fronteiras de segurança e que, às vezes, interagem com outros de formas inesperadas. Certas coisas que são assumidas como verdadeiras ou que são irrelevantes para outros programas devem ser levadas em conta em programas seguros. No UNIX, esses programas são executados pelo root ou são programas SUID de outro usuário (Bishop 1996b).

Um estudo empírico feito por Miller em 1990 identificou diversas falhas nos programas mais utilizados do UNIX, cobrindo um total de 88 programas e 7 versões do sistema. No contexto do estudo, uma falha significava ou que o programa tinha caído gerando um arquivo *core* ou que ele tinha simplesmente parado de responder (Miller 1990). O teste feito era a passagem de uma cadeia extensa de caracteres aleatórios como entrada para os programas. O programa que gerava a cadeia aleatória foi batizado de *fuzz*.

Em 1995, um estudo semelhante foi feito pelo próprio Miller junto com uma nova equipe, dessa vez com maior abrangência de programas e versões de UNIX (Miller 2000). A taxa de falhas em utilitários de versões comerciais de UNIX ainda era muito alta: de 15–43%. Além disso, muitas das falhas descobertas no estudo de 1990 foram encontradas na sua forma exata nos testes de 1995. Porém, este novo grupo de Miller não conseguiu fazer cair nenhum serviço de rede de nenhuma versão de UNIX testada.

A grande moral destes estudos é que estas falhas que os programas apresentam são potenciais furos de segurança. Se no lugar de uma string aleatória fosse passada uma string cuidadosamente construída para fazer um ataque de estouro de buffer, seriam grandes as chances de que este ataque tivesse êxito. Como veremos na seção 2, os ataques de estouro de buffer visam executar código arbitrário com as credenciais do programa atacado. Assim, se o programa for SUID *root* e o código arbitrário executa um *shell*, o atacante obtém um *shell* de *root*, ou seja, sem nenhuma restrição de segurança.

Neste artigo, procuramos dar uma breve visão sobre o mais comum ataque às aplicações – o estouro de *buffer*. Este ataque é descrito na seção 2. Na seção 3, descrevemos algumas classes de vulnerabilidades que afetam programas supostamente seguros. Na seção 4, descrevemos

algumas das ferramentas que auxiliam o programador a desenvolver código imune a ataques de estouro de *buffer*. Na seção 5, apresentamos uma conclusão seguida das referências utilizadas.

2 ATAQUE POR ESTOURO DE *BUFFER*[1]

Os ataques de estouro de *buffer* – *buffer overflow* – têm sido a forma mais comum de vulnerabilidade nos últimos 10 anos. Ainda hoje, as vulnerabilidades de estouro de *buffer* dominam a área de vulnerabilidades de penetração em redes remotas. Se essa vulnerabilidade pudesse ser eliminada, grande parte dos problemas mais sérios de segurança também o seria. O primeiro grande incidente de segurança da Internet – o *Worm*, em 1988 – utilizava a técnica do estouro de buffer no programa *fingerd*. Cowan aponta alguns números (Cowan 1999). De 13 comunicados do CERT em 1998, 9 envolviam estouro de buffer e pelo menos metade dos de 1999 também. Uma pesquisa informal na lista de discussão sobre vulnerabilidades *Bugtraq*, mostrou que 2/3 dos participantes acham que os estouros de buffer são a principal causa das vulnerabilidades. Numa visita à página de atualizações da distribuição RedHat Linux 6.2, pudemos ver que 4 de 15 atualizações corrigiam vulnerabilidades de estouro de buffer.

O principal objetivo do estouro de buffer é subverter uma função de um programa privilegiado de forma que o atacante possa controlar este programa e, se o programa for privilegiado o suficiente, controlar também a máquina onde ele está sendo executado. Normalmente o alvo do ataque é um programa com privilégios de superusuário (um programa SUID-root, por exemplo), que será conduzido a executar um código similar a "`exec(sh)`" para obter um *shell* de root. A fim de atingir este objetivo final, o atacante precisa alcançar dois sub-objetivos (Cowan 1999):

1) fazer com que o seu código arbitrário esteja no espaço de endereçamento do programa;
2) fazer o programa saltar para o endereço onde o código está.

Numa das suas formas mais comuns do ataque o atacante passa uma string cuidadosamente calculada a um programa SUID vulnerável para obter um *shell* privilegiado. A vulnerabilidade do programa é caracterizada por uma cópia ou leitura desta string sem verificação de limites do vetor destino. A Figura 1 mostra como o quadro de procedimento e as variáveis automáticas normalmente são armazenados na pilha. Se o vetor destino é uma variável local da função, ele também se localiza na pilha. Portanto, caso a string seja maior que o espaço alocado do vetor, a cópia pode sobrescrever o endereço de retorno. Essa forma do ataque é mais conhecida como *stack smashing*.

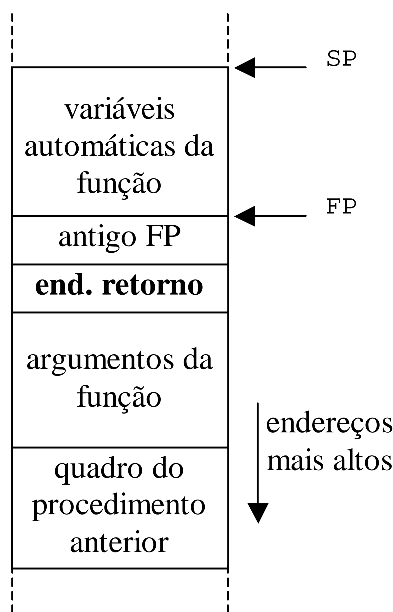

Figura 1: Situação da pilha após uma chamada de função[2].

A *string* tem seu formato descrito na Figura 2. O atacante pode utilizar-se de um programa auxiliar para gerá-la. Como podemos ver na figura, a *string* de TAM bytes que será passada como argumento para uma função é dividida em três regiões. A primeira região é preenchida com diversas instruções `NOP's`. A segunda região situa-se no centro da *string* e contém o código que dispara o *shell* – o *shellcode*. A terceira e última região é preenchida com cópias do endereço do início do código de *shell*. Ao ser passada como parâmetro

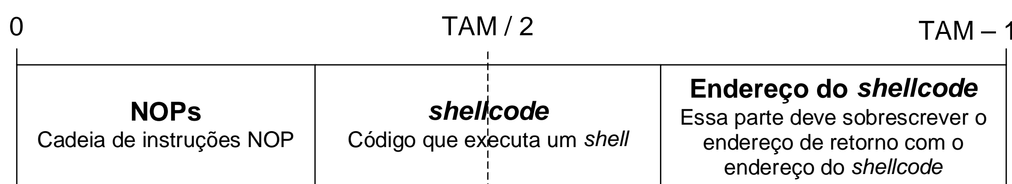

Figura 2: Formato de uma string utilizada em um ataque de estouro de *buffer*.

[1] *Buffer overflow attack.*

[2] O layout da pilha depende da arquitetura e do compilador. A figura exibe a pilha de um programa C em uma arquitetura compatível com Intel x86.

para um programa, a terceira região dessa *string* deverá sobrescrever o endereço de retorno da função atacada com o endereço passado pelo atacante que aponta para o início do código que ele quer executado. Quando a função executar uma instrução `RET`, o código do atacante passa a ser executado.

A obtenção do código de *shell* é um passo relativamente simples. AlephOne fornece em seu tutorial códigos de *shell* para diversas plataformas bem como detalhes de como efetivar o ataque (AlephOne 1996). Mas o atacante pode gerá-lo simplesmente construindo um programa que executa um *shell* e fazendo engenharia reversa no código através de um depurador. Enfim, o *shellcode* obtido será uma seqüência de bytes que representam as instruções de máquina para disparar o *shell*. É importante que este código não contenha caracteres especiais como '`\0`', '`\n`' ou `EOF`, pois geralmente a vulnerabilidade a ser atacada no programa é um procedimento de cópia de bytes que vai parar ao encontrá-los; dessa forma o buffer pode não ser estourado. É possível substituir as instruções que geram esses bytes por outras equivalentes.

As instruções `NOP` ajudam na etapa de calcular o endereço do início do *shellcode*. Se elas não precedessem *shellcode*, o atacante precisaria descobrir o endereço exato do início do mesmo, que é uma tarefa penosa e demorada. De outra forma, se o endereço usado pelo atacante for qualquer um dentro da faixa dos `NOP`'s, o código será alcançado após a execução de alguns `NOP`'s e o ataque terá sucesso.

Há casos em que o *buffer* a ser explorado é pequeno demais, ou seja, menor que o tamanho do *shellcode*. Nestes casos, o ataque deve fazer o programa saltar para o endereço de uma variável de seu próprio ambiente, a qual conterá o *shellcode*.

Algumas arquiteturas permitem que se marque o segmento de pilha como não executável. Isso evita que o código localizado na pilha seja executado, mas não evita por completo o ataque de estouro de *buffer*, pois há outras formas do ataque. Uma dessas formas consiste em fazer o endereço de retorno apontar para um código existente na aplicação que faz algo que interesse ao atacante – por exemplo, uma chamada `exec` com argumentos diferentes. Outras formas incluem forçar chamadas a outros locais como o *heap* ou o segmento de dados. No sistema operacional Linux, a pilha é executável porque em alguns momentos o próprio sistema precisa de código executável na pilha – por exemplo, para implementar sinais (Wheeler 2000).

3 OUTRAS VULNERABILIDADES

Na presente seção, apresentamos outras vulnerabilidades e relacionamos alguns cuidados para codificação de programas seguros, a exemplo de um trabalho similar na área de software crítico (ALMEIDA et al. 1999). Os exemplos referem-se à linguagem C e ao sistema operacional UNIX, pois são a linguagem e o sistema mais utilizados na rede.

Outras linguagens de programação, no entanto, merecem atenção. A linguagem *perl*, por exemplo, possui um modo mais seguro de funcionamento chamado modo "*taint*". Neste modo, o interpretador toma cuidados especiais, principalmente em relação aos dados de entrada, que previnem contra armadilhas de programação que possam causar furos de segurança. Compiladores Pascal, por exemplo, possuem uma opção que habilita a verificação de limites, que é de grande ajuda nos casos de estouro de *buffer*. Essa funcionalidade pode ser agregada à linguagem C através de *patches* para o compilador, como veremos na seção 4.

3.1 Verificação de Argumentos

Muitos *bugs* relacionados a segurança surgem porque um atacante envia um argumento inesperado ou de formato inesperado para um programa ou uma função de um programa (Garfinkel and Spafford 1996). Por isso é bom verificar todos os argumentos. Atenção extra deve ser dada aos argumentos passados para o programa na linha de comando.

É bom também garantir que os argumentos que são passados às funções do UNIX são realmente o esperado. Por exemplo, se o programador pensa que seu programa está abrindo um arquivo no diretório atual, ele pode usar a função `index` para verificar se o nome do arquivo contém um caractere barra (/). Se o nome do arquivo contém a barra, e não deveria, então o programa não deve abrir o arquivo (Garfinkel and Spafford 1996).

3.2 Erros em ponteiros e vetores

Este item tem bastante em comum com o anterior, mas merece especial atenção. Um erro comum relacionado à utilização de ponteiros é acessar os elementos de um vetor em seqüência sem verificar os seus limites. O seguinte trecho de código mostra como este erro pode ser cometido:

```
while ((c = fgetc(fin)) != '*') {
  string[i++] = c;
}
```

O trecho acima está lendo um caractere por vez do arquivo *fin* e guardando em *string*, ao mesmo tempo que incrementa o índice *i*. Esta construção "enquanto" é executada até que o caractere lido seja um asterisco '*'. É fácil notar que se o caractere asterisco estiver muito distante, o índice *i* pode alcançar o fim do espaço alocado para *string*. Isso poderá resultar na sobrescrita de outras variáveis, das informações guardadas no quadro de procedimento – como o endereço de retorno, registradores salvos, etc. – ou na violação do segmento.

Encontramos invariavelmente na literatura que o programador não deve fazer uso de funções que não verificam limites de vetor para operações em strings de tamanho arbitrário. Por exemplo, as

funções `gets()`, `strcpy()` e `strcat()` devem ser substituídas por suas similares `fgets()`, `strncpy()` e `strncat()`, respectivamente. Essas três últimas são mais seguras porque permitem a passagem de um parâmetro adicional, especificando um limite no número de bytes a serem operados.

A GNU conta com um conjunto de regras de programação para seus desenvolvedores (Stallman et al. 2000). Essas regras dizem que o programador deve alocar todas as estruturas dinamicamente, sem tentar adivinhar o tamanho máximo de alguma entrada do usuário. A equipe do FreeBSD preparou uma página na *web* que contém diversas dicas para a elaboração de um programa seguro. A primeira dica diz para não utilizar as funções `gets` e `sprintf`.

3.3 Verificação de Valores de Retorno

A não verificação de valores de retorno é um sinal de programação descuidada. Quase todas as chamadas de sistema do UNIX retornam um código de erro. Até mesmo chamadas aparentemente impossíveis de falharem, podem falhar em circunstâncias adversas. Quando as chamadas falham, a variável `errno` contém um valor que determina o motivo da falha.

Por exemplo, uma versão antiga do `su`, quando não conseguia abrir o arquivo de senhas, presumia que havia ocorrido um erro catastrófico e dava um *shell* de *root* para arrumar o sistema. Um possível ataque era abrir 19 arquivos e então realizar um "`exec su root`". Se o máximo de arquivos abertos por processo fosse 20, o ataque teria sucesso em obter um *shell* de *root*. Uma possível solução para este problema de segurança seria utilizar a variável `errno` para distinguir um erro realmente catastrófico de um ataque (Bishop 1996b).

3.4 Ambiente de Execução

A maneira mais simples de projetar um programa seguro é não assumir nada em relação ao seu ambiente e definir tudo explicitamente – sinais, *umask*, diretório atual, variáveis de ambiente. São comuns ataques onde são feitas algumas mudanças no ambiente do processo que o programador não previu.

Um furo de segurança muito conhecido envolvendo essa classe de problemas estava presente no programa `/usr/lib/preserve` (Garfinkel and Spafford 1996). Este programa, que é utilizado por editores como o *vi* e o *ex*, cria automaticamente um backup do arquivo se o usuário é desconectado inesperadamente do sistema antes de gravar as modificações no seu arquivo. O *preserve* grava as mudanças num arquivo temporário num diretório especial e depois usa o programa `/bin/mail` para enviar uma mensagem ao usuário avisando que o arquivo foi salvo.

Como os arquivos editados pelos usuários podem ser de caráter confidencial, o diretório usado por versões mais antigas do *preserve* não era acessível à maioria dos usuários do sistema. Portanto, para permitir que o *preserve* gravasse nesse diretório, estes programas foram deixados como SUID *root*.

O preserve executava o programa *mail* com a chamada `system()`, que usa o *sh* para interpretar a string que é executada. Existe uma variável de ambiente pouco conhecida chamada IFS – *internal field separator* – a qual contém os caracteres que o *sh* usa como separadores dos *tokens* para fazer o *parsing* da cadeia de caracteres. Normalmente esta variável é definida como espaço (ou TAB, nova linha etc.). Mas se esta variável incluísse o caractere barra (/), então executando o *vi* e depois executando o preserve, era possível fazer com que ele executasse um programa no diretório atual chamado *bin*; este programa era executado com privilégios de *root* (`/bin/mail` era interpretado pelo *shell* como um programa *bin* com parâmetro *mail*).

Para obter o *shell* de *root* permanente, bastava que este arquivo *bin* fosse um *script* que copiasse um *shell* para o diretório do usuário e mudasse seus atributos para SUID *root*. Na verdade era dessa forma que o *preserve* era atacado.

Alguns livros sugeriam a seguinte correção:

```
system("IFS='\n\t ';  \
  PATH=/bin:/usr/bin; \
  export IFS PATH; command");
```

Isto está errado (Bishop 1996b), pois antes o usuário poderia ter feito:

```
IFS="I$IFS";
PATH=".:$PATH"
```

E assim, o primeiro comando da chamada `system` seria a atribuição para a variável chamada `FS`, pois o caractere '`I`' passou a ser considerado um separador.

Cuidado deve ser tomado quando uma variável está definida mais de uma vez no ambiente. Qual variável será utilizada – primeira ou última – depende do sistema. Além disso, as chamadas `system` e `popen` são desaconselhadas por vários autores porque invocam o *shell* e podem ter resultados inesperados dependendo dos argumentos passados e das variáveis de ambiente (Garfinkel and Spafford 1996).

3.5 Condições de Disputa[3]

O programador deve estar ciente que seu programa não executa atomicamente. Em ambiente multitarefas, vários processos estão em execução pseudo-simultânea e isto dá margem a uma classe de ataques que exploram as condições de disputa.

A falha mais comum neste tipo de vulnerabilidade ocorre no acesso aos arquivos (Bishop 1996a). O seguinte trecho de código é uma amostra:

```
if (access(nome_arq, W_OK) == 0){
```

---

[3] *Race conditions.*

```
 if ((fd=open(nome_arq,O_WRONLY)) == NULL){
   perror(nome_arq);
   return(0);
 }
 /* ... grava no arquivo ... */
}
```

Programas SUID *root* não têm seus acessos verificados por ocasião de uma chamada `open`. Por isso, eles têm que verificar o acesso aos arquivos por si mesmos. O exemplo acima mostra o trecho de um programa desse tipo. A chamada `open`, quando executada pelo usuário *root*, nunca dará erro de permissão (afinal, *root* pode tudo...). Portanto, é necessário verificar se o usuário real pode abrir o arquivo. Isso é feito através da chamada `access`, que devolve 0 em caso positivo.

O grande defeito desse trecho de código é que há uma janela de vulnerabilidade entre a chamada `access` e a chamada `open`. O atacante pode substituir o arquivo original por um *link* para qualquer arquivo que deseje ler e não tenha acesso – ou seja, mudar o conteúdo que é apontado pelo nome de arquivo contido na variável `nome_arq`. A chamada access será efetuada sobre um arquivo enquanto a chamada open será efetuada em outro arquivo. Essa falha faz parte de uma classe de falhas chamada TOCTTOU – *Time Of Check To Time Of Use* (Bishop 1996a).

Para evitar esse tipo de vulnerabilidade, o programador deve procurar utilizar as funções que recebem como argumento o descritor do arquivo em vez das que recebem o nome do arquivo.

3.6 Ligação Dinâmica

Na ligação dinâmica, as funções de biblioteca não estão na memória no momento do início da execução do programa. Em vez disso, há um chamado *stub*. Quando uma função que não está na memória é chamada, o *stub* se encarrega de procurar a função e carregá-la, para que o fluxo possa ser então desviado para ela. Muitos sistemas UNIX, no momento de carregar a rotina para a memória, procuram a mesma em diretórios que estão guardados em variáveis de ambiente, tais como LD_LIBRARY_PATH ou LD_PRELOAD. Dessa forma, um programa SUID que utiliza ligação dinâmica tem partes controladas pelo usuário.

Um possível ataque a essa vulnerabilidade é como segue (Bishop 1996b). Seja `fgets` uma função a ser carregada dinamicamente que é utilizada por um programa SUID. O atacante pode criar uma biblioteca dinâmica no seu diretório pessoal contendo a seguinte rotina `fgets` substitutiva:

```
fgets(char *buf, int n, FILE *fp)
{
  execl("/bin/sh", "-sh", 0);
}
```

E colocá-la numa biblioteca dentro do diretório atual. Ao executar:

```
$ LD_PRELOAD=.:$LD_PRELOAD
$ progsuid
#
```

O diretório corrente é colocado no início do caminho de busca de bibliotecas; assim, a rotina `fgets` encontrada pelo *stub* será a rotina do atacante, que dispara um *shell* com as permissões do dono do programa.

Na verdade, os programas SUID desconsideram a variável LD_PRELOAD e utilizam as bibliotecas localizadas em diretórios confiáveis. Porém, um processo filho invocado por um programa SUID herda seu ambiente e irá utilizar a variável LD_PRELOAD.

Sugere-se que em programas SUID, as bibliotecas sejam ligadas estaticamente. Os compiladores geralmente fornecem uma opção para isso.

3.7 Chamada de Subprocessos

É importante que os processos SUID *root* abram mão de seus privilégios antes de executar outros programas. Isto porque as credenciais do processo pai são herdadas pelo processo filho.

Bishop relata um incidente de segurança envolvendo alguns jogos populares que estavam presentes no sistema (Bishop 1996b). Os jogos precisavam alterar o placar geral (que qualquer usuário, por seu desempenho no jogo pode indiretamente alterar) e por isso eram SUID *root*. Foi descoberto que a UID efetiva não era redefinida quando um *shell* era disparado pelo jogo. Portanto, era possível rodar um jogo que mantinha um arquivo de placar e chamar um *subshell* – este teria privilégios de *root*.

A regra básica de segurança de computadores é minimizar os danos resultantes de um ataque. Para este problema dos jogos SUID *root*, uma das soluções é ser mais restritivo no que tange à escolha do usuário e do grupo. Em vez de *root*, seria mais seguro ter um usuário chamado *games* que teria a posse dos arquivos de placar. Se assim o fosse, o ataque dos jogos teria obtido os privilégios do usuário *games* apenas, que são bem mais restritos.

4 FERRAMENTAS

Diversas ferramentas existem para auxiliar o programador a desenvolver código seguro, com relação a estouro de buffer, que é a principal vulnerabilidade no nível de aplicação. Nesta seção procuramos apresentar algumas delas. Podemos dividir as ferramentas de verificação de código em estáticas e dinâmicas. As estáticas analisam o código sem que o mesmo esteja sendo executado enquanto que as dinâmicas trabalham em tempo de execução. Além das ferramentas aqui descritas, existem várias outras relacionadas na página http://lclint.cs.virginia.edu/links.html.

4.1 BoundsChecking

Esta ferramenta de verificação dinâmica foi inicialmente desenvolvida por Richard Jones e Paul Kelly (Jones and Kelly 1997). Atualmente ela é mantida por Herman Ten Brugge e distribuída sob os termos da GPL. Mais informações podem ser obtidas na página http://web.inter.nl.net/hcc/Haj.Ten.Brugge/.

Trata-se de pequenas alterações feitas no gcc para adicionar código de verificação nas operações aritméticas e no uso de ponteiros, mantendo uma tabela de regiões alocadas conhecidas. Os autores realizaram testes de desempenho verificando que o tempo de execução de um programa compilado com *BoundsChecking* é praticamente inafetado em relação à versão normal. Porém, isso depende muito do modo como o programa foi escrito.

4.2 StackGuard

O *StackGuard* é uma extensão ao compilador que melhora o código executável produzido pelo compilador de forma que ele possa detectar ataques de estouro de buffer contra a pilha. O efeito é transparente ao funcionamento normal dos programas. Esta ferramenta foi proposta por Cowan (Cowan 1998).

O *StackGuard* funciona basicamente inserindo uma palavra especial (*canary word*) na pilha, entre o endereço de retorno e as variáveis locais. Grande parte dos ataques de estouro de buffer se dá através da alteração do endereço de retorno que está na pilha. O *StackGuard* consegue, através dessa palavra que foi inserida na pilha, detectar mudanças no endereço de retorno.

Recentemente a Wirex lançou uma distribuição de Linux chamada Immunix OS 6.2. Ela é baseada diretamente no RedHat Linux 6.2, mas todos os programas com código-fonte em C disponível foram recompilados com o compilador *StackGuard*.

Maiores informações sobre o *StackGuard* podem ser obtidas no site http://www.immunix.org/.

4.3 StackShield

O *StackShield* possui o mesmo objetivo do *StackGuard*: detectar escritas indevidas na pilha. Eles diferem na forma como essa escrita é detectada e tratada. O *StackShield* atua como um processador de programas-fonte *assembly*, é suportado pelos *front-ends* do GCC e é distribuído sob a licença GPL.

O *StackShield* modifica o prólogo e o epílogo das funções nos programas compilados pelo GCC. No prólogo das funções ele faz uma cópia do endereço de retorno para um espaço que não pode ser atingido e compara, no epílogo da função, se os dois valores são diferentes. Se forem diferentes, o endereço de retorno foi modificado e o endereço de retorno original é restaurado para que o programa siga executando. Porém, como geralmente os ataques de estouro de buffer alteram grande parte do estado do programa, as chances são grandes do mesmo vir a falhar.

No fim de agosto de 1999 ocorreu uma discussão na lista BUGTRAQ envolvendo esta ferramenta e sua similar. O autor do *StackGuard* – Cowan – questionou a autenticidade de uma informação[4] contida na página do *StackShield*, que dizia que o *StackShield* é mais seguro que o *StackGuard*. Cowan fez uma comparação entre os dois sistemas e contestou a afirmação do autor do *StackShield*, dizendo ainda que em determinado modo de operação o *StackGuard* apresentava uma performance melhor que a do *StackShield*. No entanto, uma das vantagens do *StackShield* em relação ao outro, é que ele pode ser utilizado com o *gdb*. O *StackGuard*, por sua vez, só irá funcionar com um *gdb* alterado, que conheça o seu formato de quadro de procedimento.

Maiores informações sobre o *StackShield* podem ser obtidas no site http://www.angelfire.com/sk/stackshield.

4.4 LCLint

Logo após o nascimento da linguagem C, sem prototipação de funções, era de comum acordo que a depuração de programas era uma tarefa difícil. Dessa forma, foi criada uma ferramenta chamada *lint* capaz de fazer diversas verificações estáticas do código. Este programa foi escrito por S. C. Johnson no início dos anos 70 e foi a primeira ferramenta de validação estática de código.

Com a criação do C padrão ANSI. algumas verificações do *lint* tornaram-se supérfluas. John Guttag e Jim Horning criaram a ferramenta *LCLint*, muito semelhante ao *lint*. Os comandos de *lint* podem ser emulados por *LCLint*, e os erros detectados por *lint* tais como: declarações não utilizadas, inconsistências de tipos, código inatingível, utilização antes da definição, prováveis laços infinitos e casos de *fall-through*, valores de retorno ignorados e caminhos de execução sem retorno, são também detectadas pelo *LCLint*.

A idéia básica por trás do *LCLint* é executá-lo antes de compilar o código. Assim o *LCLint* vai procurar os possíveis erros no programa informando o usuário de seus achados. O *LCLint* possui vários modos de análise. No modo mais básico, ele atua como um verificador estático, sem tomar conhecimento da semântica do programa e é recomendado para depuração de código escrito por outrem. Em outro modo de operação, o *LCLint* interpreta anotações passadas dentro de comentários do programa, nas quais o programador pode informar o significado do seu programa. Este último modo é bem mais poderoso que o primeiro e nele o *LCLint* pode deduzir se o programa faz o que o programador realmente quer que ele faça.

O código fonte de *LCLint* pode ser encontrado em http://lclint.cs.virginia.edu/download.html.

---

[4] No momento da escrita deste artigo, essa mesma informação ainda consta na página.

Binários para Linux, Windows e FreeBSD também estão disponíveis.

4.5 Purify

O *Purify*, da *Rational*, adiciona instruções adicionais no código objeto antes de cada operação de LOAD e de STORE. Das diferenças entre o *Purify* e as outras ferramentas comentadas até o momento, é que ela atua no código objeto e não necessita do código fonte para que seja aplicada. Portanto, o programador pode com o *Purify*, proteger seu programa de *bugs* que aparecem até mesmo em bibliotecas fornecidas por terceiros.

O *Purify* apresenta facilidade de uso e os binários gerados por ele podem ser manipulados por um depurador comum. Além disso, ele oferece a opção de uma interface gráfica onde podem ser exibidas as mensagens de erro.

Dentre os muitos erros que podem ser identificados pelo código gerado com o *Purify*, destacamos os seguintes:

- variáveis locais não inicializadas;
- memória alocada dinamicamente não inicializada, utilização de locais de memória já liberados;
- escrita ou leitura além dos limites de um vetor;
- erros de estouro de pilha;

Maiores informações sobre o Purify podem ser obtidas no site http://www.rational.com/.

5 CONCLUSÕES

A maior parte dos ataques de segurança atuais baseia-se no estouro de buffer, que é uma técnica bem conhecida e relativamente antiga. Isso mostra que os programadores ainda não se adaptaram para escrever programas realmente seguros. Citando o que diz na página intitulada "Security Code Guidelines" da Sun (Sun 2000):

"*uma corrente só pode ser tão forte quanto o mais fraco de seus elos*"

O elo mais fraco no caso de linguagens projetadas visando a segurança – como Java – é o programador. A solução neste caso é o treinamento e a informação, apenas. No caso da linguagem C, cujos princípios de projeto não visavam a segurança do código gerado, além do treinamento, é desejável que se utilize ferramentas de verificação de código, como as apresentadas na seção 4, e *patches* que agregam funcionalidades de segurança ao compilador. Podemos encontrar na rede, farta documentação com regras e cuidados para o desenvolvimento de código seguro. A maioria refere-se à linguagem C e ao ambiente UNIX, mas linguagens script como CGI e Perl, que também são bastante utilizadas em servidores *Web*, estão disponíveis.

REFERÊNCIAS BIBLIOGRÁFICAS

AL-HERBISH, Thamer. Secure Unix Programming FAQ. Version 0.5, May 16, 1999. Cópia principal em http://www.whitefang.com/sup/

ALMEIDA Jr., J. R.; CAMARGO Jr., J. B.; BASSETO, B. A.; CUNHA, R. S.; PAZ, S. M. *Uma Lista de Inspeção para a Análise de Software Crítico*. In: SIMPÓSIO SOBRE SEGURANÇA EM INFORMÁTICA, 99, São José dos Campos. *Anais*... São José dos Campos: 1999. P. 67-74.

ALEPHONE. *Smashing the Stack for Fun and Profit*. Phrack Magazine. Volume 7, edição 49, arquivo 14 de 16, 199?.

BISHOP, M. and DILGER, M. Checking for Race Conditions in File Accesses. Computing Systems 9(2), pp. 131-152, Primavera de 1996 (a).

BISHOP, Matt. *UNIX Security: Security in Programming*. SANS '96, Washington DC, Maio de 1996 (b).

BISHOP, Matt. *Writing Safe SUID Programs*. Network Security 1997, New Orleans, LA, Outubro de 1997.

COWAN, Crispin et al. *StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks*. 7th USENIX Security Conference, pp. 63–77, San Antonio, TX, Janeiro de 1998.

COWAN, Crispin et al. *Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade*. DARPA Information Survivability Conference and Expo (DISCEX), 25–27 de Janeiro de 2000.

GARFINKEL, S. and SPAFFORD, G. *Practical Unix & Internet Security*. 2ª Edição, O'Reilly, 1996.

JONES, R. W. M. and KELLY, P. H. J. Backwards-compatible bounds checking for arrays and pointers in C programs. AADEBUG'97, Linköping, Suécia, 1997.

MILLER, B. P. et al. *An Empirical Study of the Reliability of UNIX Utilities*. Communications of ACM. Volume 33, number 12, December 1990, 32-44.

MILLER, B. P. et al. *Fuzz Revisited: A Re-examination of the Reliability of UNIX Utilities and Services*. 18 de fevereiro de 2000. ftp://grilled.cs.wisc.edu/technical_papers/fuzz-revisited.ps.Z

SPAFFORD, Eugene. *Crisis and Aftermath*. Communications of ACM. Volume 32, número 6, Junho de 1989, p. 678-687.

STALLMAN, Richard et al. *GNU Coding Standards*. Free Software Foundation Inc. Última atualização em 27/06/2000. Documento on-line. URL: http://www.gnu.org/prep/ standards_toc.html

STEVENS, W. Richard. *Advanced Programming in the UNIX Environment*. Addison-Wesley, 1992.

SUN Microsystems Inc. *Security Code Guidelines*. Documento on-line. URL: http://java.sun.com/ security/seccodeguide.html. Versão de 2 de fevereiro de 2000.

WHEELER, David A. *Secure Programming for Linux and UNIX HOWTO*. Versão 2.20, julho de 2000. URL: http://www.dwheeler.com/secure-programs.