MINISTÉRIO DA CIÊNCIA E TECNOLOGIA
INSTITUTO NACIONAL DE PESQUISAS ESPACIAIS

Programação Segura: Uma Introdução à Auditoria de Códigos

Luiz Gustavo C. Barbato, Luiz Otávio Duarte
{lgbarbato,duarte}@lac.inpe.br
RESSIN - Redes e Segurança de Sistemas de Informação
LAC - Laboratório Associado de Computação e Matemática Aplicada
INPE - Instituto Nacional de Pesquisas Espaciais

Antonio Montes
antonio.montes@cenpra.gov.br
CenPRA - Centro de Pesquisas Renato Archer

Introdução

- Quem somos?
  - Pós-Graduação em Computação Aplicada

    `http://www.lac.inpe.br/cap`
- Quais as linhas de pesquisas em segurança?
  - SDI, ICP, Forense, *Honeynet*, *Software*,...

    `http://www.lac.inpe.br/security`
- Por que Segurança de *Software*?
  - Muitos problemas com poucas soluções
  - Soluções difíceis de serem implantadas
  - **DESAFIO**

Roteiro

- Objetivo
- Vulnerabilidades
- Problemas na Codificação
- Auditoria de Códigos
- Ferramentas de Auxílio à Auditoria
- Problemas nas Ferramentas
- Conclusões

Objetivo

O objetivo deste trabalho é mostrar alguns dos problemas atualmente encontrados na codificação dos sistemas, como estouro de buffers, condições de corrida e validações de entrada, assim como mostrar algumas ferramentas que auxiliam à auditoria de códigos, como Flawfinder, ITS4, PScan e RATS, na procura por estes problemas.

Vulnerabilidades

- Vulnerabilidades são condições de fraqueza
- Tipos de vulnerabilidades:
  - Físicas
  - *Hardware*
  - Naturais
  - Humanas
  - *Software*

Vulnerabilidades em *Software*

- Problemas de configuração
  - Desconhecimento do sistema
- Sistemas mal projetados
  - Requisito não funcional
- Erros na codificação
  - Falta de conhecimento

Dados Alarmantes

- **Dados do ICAT/NIST** mostram que no **ano de 2004** uma grande porcentagem dos problemas de segurança encontradas em sistemas é devida a **má codificação** dos mesmos.
- Do total de vulnerabilidades encontradas, pelo menos **74%** eram decorrentes de **falhas de codificação**.
- Do início do ano **até agosto**, *52%* dos erros eram devidos à falhas na **validação de entradas**, que envolvem problemas de **estouro de *buffers*.**

Por que os problemas ocorrem?

- **Email enviado por Aleph One em dezembro de 1998 para a *Bugtraq***

  `http://seclists.org/bugtraq/1998/Dec/0062.html`

- A maioria das universidades não se preocupam com disciplinas voltadas a segurança de computadores;
- Livros de programação não ensinam técnicas de programação segura;
- A linguagem C é insegura;
- Programadores não pensam em ambientes ”multi-usuários“;

Por que os problemas ocorrem? (cont.)

- Programadores são humanos. Humanos são preguiçosos;
- A maioria dos programadores não são bons programadores;
- A maioria dos programadores não trabalham com segurança;
- Consumidores não se preocupam com segurança;
- Custo extra no tempo de desenvolvimento;

Problemas na Codificação

Dentre os vários problemas foram selecionados 3, os quais serão alvo de detecção ou não das ferramentas

- Estouro de *Buffer*
- Condição de corrida
- Validações de Entradas

Estouro de *Buffer*

- Os estouros de *buffers* são problemas causados devido a não validação do tamanho da memória utilizada, excedendo em certas circunstâncias, a sua capacidade de armazenamento.
- ***Heap Overflow***: Estouro de áreas alocadas para variáveis globais, variáveis alocadas dinamicamente, ...
- ***Stack Overflow***: Estouro de áreas alocadas para variáveis locais, parâmetros de funções, ...

Estouro de *Buffer* (cont.)

Programa de Exemplo:

```
$ cat bo.c

int main (int argc, char ** argv) {

  char buffer[16];
  strcpy (buffer, argv[1]);
  printf ("%s\n", buffer);

  return 0;

}
```

Estouro de *buffer* (cont.)

Execução do programa:

```
$ ./bo teste-overflow
teste-overflow
```

Teste de vulnerabilidade:

```
$ ./bo teste-overflowwwwwwwwwwwwww
teste-overflowwwwwwwwwwwwww
Segmentation fault
```

Condições de Corrida

- As condições de corrida ocorrem em ambientes que suportam multiprogramação.
- Este problema acontece quando dois ou mais processos acessam os mesmos recursos "simultaneamente".
- Um recurso pode ser modificado, intencionalmente ou não, por um processo, e será requisitado por um segundo, fazendo que este se comporte de maneira não esperada.

Condições de Corrida (cont.)

Programa de Exemplo:

```
$ cat rc.c

#include <stdio.h>
int main () {

  FILE * fd = fopen ("/tmp/arquivo-temporario", "w");
  fputs ("teste de condicao de corrida\n", fd);
  fclose (fd);

  return 0;

}
```

Condições de Corrida (cont.)

Execução do programa:

```
$ ./rc
$ cat /tmp/arquivo-temporario
teste de condicao de corrida
```

Teste de vulnerabilidade:

```
$ ln -s /tmp/arquivo-indevido /tmp/arquivo-temporario
$ ./rc
$ cat /tmp/arquivo-temporario
teste de condicao de corrida
$ cat /tmp/arquivo-indevido
teste de condicao de corrida
```

Validações de Entradas

- Este problema consiste da não verificação dos valores de entrada nos programas, gerando assim, situações inesperadas como execução de códigos indevidos, estouros de *buffers*, execução de comandos SQL não permitidos, dentre outras.
- A entrada de dados em um programa pode ser feita de várias maneiras como parâmetros de execução do programa, leituras de teclado e arquivos, através de comunicação inter-processos (memória compartilhada, *pipe*,...) e via rede (*sockets*), etc.

Validações de Entradas (cont.)

Programa de Exemplo:

```
$ cat ve.c

int main (int argc, char ** argv) {

  char tam_dir[32];
  snprintf (tam_dir, 32, "du -sh %s", argv[1]);
  system(tam_dir);

  return 0;

}
```

Validações de Entradas (cont.)

Execução do programa:

```
$ ./ve /tmp
13M     /tmp
```

Teste de vulnerabilidade:

```
$ ./ve "/tmp;date"
13M     /tmp
Wed Sep 29 19:49:13 EST 2004
```

Último Código de Exemplo

```
$ cat io.c

int main(int argc, char *argv[])
{
  char buff[16];
  int i=0;
  int aux=strlen(argv[1]);

  while(aux >= 0) buff[i++] = argv[1][--aux];
  buff[i]='\0';

  printf("%s\n",buff);
  return(0);
}
```

Execução do Último Código de Exemplo

Execução do programa:

```
$ ./io teste-overflow
wolfrevo-etset
```

Teste de vulnerabilidade:

```
$ ./io teste-overflowwwwwwwwwwwwww
wwwwwwwwwwwwwwwolfrevo-etset
Segmentation fault
```

Auditoria de Códigos

Pode ser realizada de duas maneiras:

- Dinâmica
- **Estática**
  - Por meio da análise de todo o código fonte;
  - Por meio do rastreamento dos pontos de iteração com outros processos ou usuários;
  - Por meio de buscas por assinaturas de funções.
  - ...

Ferramentas de Auxilio à Auditoria

Existe uma gama de ferramentas capazes de auxiliar a análise estática de códigos. As mais conhecidas e difundidas são:

- Flawfinder
- ITS4
- RATS
- PScan

Baseada na lista do *Sardonix*. `<http://www.sardonix.org>`

Flawfinder

- É uma ferramenta escrita em *Python* desenvolvida para analisar códigos fontes da linguagem “C”.
- Identifica os seguintes problemas:
  - *Buffer Overflow* em funções conhecidas como *“strcpy();”*, *“strcat();”*, *“gets();”*, *“sptrintf();”* e funções da família *“scanf();”*;
  - *Format Strings* em funções das famílias *“[v][f]printf();”*, *“[v]snprinf();”* e *“syslog();”*;
  - *Race Conditions* em funções como: *“access();”*, *“chown();”*, *“chgrp();”*, *“chmod();”*, *“tmpfile();”*, *“tmpnam();”* e *“mktemp();”*.

ITS4

- Realiza análises sobre trechos de códigos *“C/C++”*;
- Trabalha com um banco de funções potencialmente inseguras, conseguindo identificar casos de *buffer overflow*, *race condition*, *Format String*;
- Permite que o relatório gerado seja ordenado.
- Permite que trechos de códigos sejam excluídos da verificação:

```
strcpy(dst, src);  /* ITS4: ignore */
```

RATS

- É uma ferramenta que tem a capacidade de trabalhar com uma gama de linguagens como: C, *Python*, *php*, *perl* e etc..
- Objetiva encontrar problemas de *Buffer Overflow* e *Race Condition*;
- Trabalha com bancos de dados XML o que faz com que seja necessário que a biblioteca *Expat* esteja instalada no sistema.

PScan

- É uma ferramenta que tem por objetivo encontrar problemas de *format string* em funções da família *printf*;
- Permite que definições adicionais sejam inseridas;
- Trabalha com a seguinte regra:

```
IF the last parameter of the function is the format string,
AND the format string is NOT a static string,
THEN complain.
```

Ferramentas X Exemplos

Ferramenta	bo.c	rc.c	ve.c	io.c	fs.c
Flawfinder	*/+	*	*/+	-/+	*/+
ITS4	*/+	*	*	-/+	*/+
RATS	*/+	-	*/+	-/+	*/+
PScan	-	-	-	-	*

	Legenda
*	A vulnerabilidade foi encontrada;
-	A vulnerabilidade não foi encontrada;
+	Mais informações, por vezes não vulneráveis.

O que as ferramentas podem induzir?

- O analista inexperiente pode ficar com uma falsa sensação de segurança. Isto pois, nem todas as reais vulnerabilidades podem ser encontradas e nem todas as vulnerabilidades encontradas são de fato vulnerabilidades.
- Entretanto, estas ferramentas podem auxiliar na conscientização de programadores, através de uma análise preliminar do programa.

Conclusão

- As ferramentas atualmente disponíveis ainda requerem um alto grau de conhecimento do analista.[2] Além disso, segundo Jon Heffley et al. [10], na maior parte das vezes de 67% a 75% do trabalho de auditoria completa do código ainda é requerida;
- A análise estática de código, com o auxilio de ferramentas, ainda é dispendiosa devido ao grande número de falsos positivos;
- Entretanto, à medida que novas ferramentas são desenvolvidas, as atuais se tornem mais maduras e a pesquisa nesta área aumente; o número de falsos positivos e de falhas não identificadas tendem a diminuir, tornando assim estas ferramentas mais confiáveis e efetivas.

Referências

[1] Greg Hoglund and Gary McGraw. *Exploiting Software: How to Break Code*. Addison-Wesley, 1st edition, February 2004. ISBN 0-201-76895-8

[2] John Viega and Gary McGraw. *Building Secure Software: Howto Avoid Security Problems the Right Way*. Addison-Wesley, 1st edition, September 2001. ISBN 0-201-72152-X.

[3] Mark G. Graff and Kenneth R. Van Wyk. *Secure Coding: Principles and Pratices*. O'Reilly & Associates, 1st edition, Jully 2003. ISBN 0-596-00242-4.

[4] John Viega and Matt Messier *Secure Programming Cookbook for C and C++* O'Reilly & Associates, 1st edition, Jully 2003. ISBN 0-596-00394-3.

Referências (cont.)

[5] Michael Howard and David C. LeBlanc. *Writing Secure Code* Microsoft Press, 2nd edition, December 2003. ISBN 0-735-61722-8.

[6] Ross J. Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems* Willey, 1st edition, January 2001. ISBN 0-471-38922-6.

[7] James A. Whittaker and Herbert H. Thompson. *How to Break Software Security*. Pearson Education, 1st edition, May 2003. ISBN 0-321-19433-0

[8] Brian W. Kernighan and Rob Pike *The Practice of Programming*. Addison-Wesley, 1st edition, February 1999. ISBN 0-201-61586-X

Referências (cont.)

[9] David A. Wheeler *Secure Programming for Linux and Unix HOWTO*. March 2003. Disponível on-line em: `http://www.dwheeler.com/secure-programs` verificado em setembro de 2004.

[10] Jon Heffley and Pascal Meunier. *Can Source Code Auditing Software Identify Common Vulnerabilities and Be Used to Evaluate Software Security?*. Proceedings of the 37th Hawaii International Conference on System Sciences, 2004

[11] David A. Wheeler. *Flawfinder*. Disponível on-line em: `http://www.dwheeler/flawfinder` verificado em setembro de 2004.

Referências (cont.)

[12] *RATS: Rough Auditing Tool for Security* Disponível on-line em: `http://www.securesw.com/download_rats.html` verificado em setembro de 2004.

[13] Cigital. *ITS4: Software Security Tool*. Disponível on-line em: `http://www.cigital.com/its4/` verificado em setembro de 2004.

[14] Alan DeKok. *PScan: A limited problem scanner for C source files*. Disponível on-line em: `http://www.striker.ottawa.on.ca/~aland/pscan/` verificado em agosto de 2004.

Obrigado

Luiz Gustavo Cunha Barbato

<lgbarbato@lac.inpe.br>

Luiz Otávio Duarte

<duarte@lac.inpe.br>

Antonio Montes

<antonio.montes@cenpra.gov.br>

Exemplo fs.c

```
$ cat fs.c

int main(int argc,char *argv[]) {
  char msg[16];

  if (strlen(argv[1]) < 16){
    sprintf(msg,argv[1]);
    printf("%s\n",msg);
  } else
    printf("Tamanho de entrada inválido.\n");

  return 0;

}
```

Saída Flawfinder para bo.c

```
$ flawfinder bo.c

Flawfinder version 1.26, (C) 2001-2004 David A. Wheeler.
Number of dangerous functions in C/C++ ruleset: 158
Examining bo.c
bo.c:6:  [4] (buffer) strcpy:
  Does not check for buffer overflows when copying to destination.
  Consider using strncpy or strlcpy (warning, strncpy is easily misused).
bo.c:5:  [2] (buffer) char:
  Statically-sized arrays can be overflowed. Perform bounds checking,
  use functions that limit length, or ensure that the size is larger than
  the maximum possible length.

Hits = 2
Lines analyzed = 11 in 0.57 seconds (150 lines/second)
Physical Source Lines of Code (SLOC) = 7
Hits@level = [0]   0 [1]   0 [2]   1 [3]   0 [4]   1 [5]   0
Hits@level+ = [0+]   2 [1+]   2 [2+]   2 [3+]   1 [4+]   1 [5+]   0
Hits/KSLOC@level+ = [0+] 285.714 [1+] 285.714 [2+] 285.714 [3+] 142.857 [4+] 142.857 [5+]   0
Minimum risk level = 1
Not every hit is necessarily a security vulnerability.
There may be other security vulnerabilities; review your code!
```

Saída Flawfinder para rc.c

```
$ flawfinder rc.c

Flawfinder version 1.26, (C) 2001-2004 David A. Wheeler.
Number of dangerous functions in C/C++ ruleset: 158
Examining rc.c
rc.c:6:  [2] (misc) fopen:
  Check when opening files - can an attacker redirect it (via symlinks),
  force the opening of special file type (e.g., device files), move
  things around to create a race condition, control its ancestors, or change
  its contents?.

Hits = 1
Lines analyzed = 12 in 0.59 seconds (130 lines/second)
Physical Source Lines of Code (SLOC) = 8
Hits@level = [0]   0 [1]   0 [2]   1 [3]   0 [4]   0 [5]   0
Hits@level+ = [0+]   1 [1+]   1 [2+]   1 [3+]   0 [4+]   0 [5+]   0
Hits/KSLOC@level+ = [0+] 125 [1+] 125 [2+] 125 [3+]   0 [4+]   0 [5+]   0
Minimum risk level = 1
Not every hit is necessarily a security vulnerability.
There may be other security vulnerabilities; review your code!
```

Saída Flawfinder para ve.c

```
$ flawfinder ve.c

Flawfinder version 1.26, (C) 2001-2004 David A. Wheeler.
Number of dangerous functions in C/C++ ruleset: 158
Examining ve.c
ve.c:7:  [4] (shell) system:
  This causes a new program to execute and is difficult to use safely.
  try using a library call that implements the same functionality if
  available.
ve.c:5:  [2] (buffer) char:
  Statically-sized arrays can be overflowed. Perform bounds checking,
  use functions that limit length, or ensure that the size is larger than
  the maximum possible length.
ve.c:6:  [1] (port) snprintf:
  On some very old systems, snprintf is incorrectly implemented and
  permits buffer overflows; there are also incompatible standard definitions
  of it. Check it during installation, or use something else.

Hits = 3
Lines analyzed = 11 in 0.57 seconds (165 lines/second)
Physical Source Lines of Code (SLOC) = 7
Hits@level = [0]   0 [1]   1 [2]   1 [3]   0 [4]   1 [5]   0
Hits@level+ = [0+]   3 [1+]   3 [2+]   2 [3+]   1 [4+]   1 [5+]   0
Hits/KSLOC@level+ = [0+] 428.571 [1+] 428.571 [2+] 285.714 [3+] 142.857 [4+] 142.857 [5+]   0
Minimum risk level = 1
Not every hit is necessarily a security vulnerability.
There may be other security vulnerabilities; review your code!
```

Saída Flawfinder para io.c

```
$ flawfinder io.c

Flawfinder version 1.26, (C) 2001-2004 David A. Wheeler.
Number of dangerous functions in C/C++ ruleset: 158
Examining io.c
io.c:5:  [2] (buffer) char:
  Statically-sized arrays can be overflowed. Perform bounds checking,
  use functions that limit length, or ensure that the size is larger than
  the maximum possible length.
io.c:7:  [1] (buffer) strlen:
  Does not handle strings that are not \0-terminated (it could cause a
  crash if unprotected).

Hits = 2
Lines analyzed = 14 in 0.57 seconds (206 lines/second)
Physical Source Lines of Code (SLOC) = 11
Hits@level = [0]   0 [1]   1 [2]   1 [3]   0 [4]   0 [5]   0
Hits@level+ = [0+]   2 [1+]   2 [2+]   1 [3+]   0 [4+]   0 [5+]   0
Hits/KSLOC@level+ = [0+] 181.818 [1+] 181.818 [2+] 90.9091 [3+]   0 [4+]   0 [5+]   0
Minimum risk level = 1
Not every hit is necessarily a security vulnerability.
There may be other security vulnerabilities; review your code!
```

Saída Flawfinder para fs.c

```
$ flawfinder fs.c

Flawfinder version 1.26, (C) 2001-2004 David A. Wheeler.
Number of dangerous functions in C/C++ ruleset: 158
Examining fs.c
fs.c:5:  [4] (format) sprintf:
  Potential format string problem. Make format string constant.
fs.c:2:  [2] (buffer) char:
  Statically-sized arrays can be overflowed. Perform bounds checking,
  use functions that limit length, or ensure that the size is larger than
  the maximum possible length.
fs.c:4:  [1] (buffer) strlen:
  Does not handle strings that are not \0-terminated (it could cause a
  crash if unprotected).

Hits = 3
Lines analyzed = 12 in 0.56 seconds (206 lines/second)
Physical Source Lines of Code (SLOC) = 9
Hits@level = [0]   0 [1]   1 [2]   1 [3]   0 [4]   1 [5]   0
Hits@level+ = [0+]   3 [1+]   3 [2+]   2 [3+]   1 [4+]   1 [5+]   0
Hits/KSLOC@level+ = [0+] 333.333 [1+] 333.333 [2+] 222.222 [3+] 111.111 [4+] 111.111 [5+]   0
Minimum risk level = 1
Not every hit is necessarily a security vulnerability.
There may be other security vulnerabilities; review your code!
```

Saída ITS4 para bo.c

```
$ its4 bo.c

bo.c:7:(Urgent) printf
Non-constant format strings can often be attacked.
Use a constant format string.
----------------
bo.c:6:(Very Risky) strcpy
This function is high risk for buffer overflows
Use strncpy instead.
----------------
```

Saída ITS4 para rc.c

```
$ its4 rc.c

rc.c:6:(Risky) fopen
Can be involved in a race condition if you open things after a poor check. For
example, don't check to see if something is not a symbolic link before opening
it.  Open it, then check bt querying the resulting object.  Don't run tests on
symbolic file names...
Perform all checks AFTER the open, and based on the returned object, not a
symbolic name.
----------------
```

Saída ITS4 para ve.c

```
$ its4 ve.c

ve.c:7:(Urgent) system
Easy to run arbitrary commands through env vars. Also, potential TOCTOU
problems.
Use fork + execve instead.
----------------
```

Saída ITS4 para io.c

```
$ its4 io.c

io.c:12:(Urgent) printf
Non-constant format strings can often be attacked.
Use a constant format string.
----------------
```

Saída ITS4 para fs.c

```
$ its4 fs.c

fs.c:6:(Urgent) printf
fs.c:8:(Urgent) printf
Non-constant format strings can often be attacked.
Use a constant format string.
----------------
fs.c:5:(Urgent) sprintf
Non-constant format strings can often be attacked.
Use a constant format string.
----------------
```

Saída RATS para bo.c

```
$ rats -d rats-c.xml bo.c

Entries in c database: 310
Analyzing bo.c
bo.c:5: High: fixed size local buffer
Extra care should be taken to ensure that character arrays that are allocated
on the stack are used safely.  They are prime targets for buffer overflow
attacks.

bo.c:6: High: strcpy
Check to be sure that argument 2 passed to this function call will not copy
more data than can be handled, resulting in a buffer overflow.

Total lines analyzed: 12
Total time 0.000301 seconds
39867 lines per second
```

Saída RATS para rc.c

```
$ rats -d rats-c.xml rc.c

Entries in c database: 310
Analyzing rc.c
Total lines analyzed: 13
Total time 0.001793 seconds
7250 lines per second
```

Saída RATS para ve.c

```
$ rats -d rats-c.xml ve.c

Entries in c database: 310
Analyzing ve.c
ve.c:5: High: fixed size local buffer
Extra care should be taken to ensure that character arrays that are allocated
on the stack are used safely.  They are prime targets for buffer overflow
attacks.

ve.c:7: High: system
Argument 1 to this function call should be checked to ensure that it does not
come from an untrusted source without first verifying that it contains nothing
dangerous.

Total lines analyzed: 12
Total time 0.000931 seconds
12889 lines per second
```

Saída RATS para io.c

```
$ rats -d rats-c.xml io.c

Entries in c database: 310
Analyzing io.c
io.c:5: High: fixed size local buffer
Extra care should be taken to ensure that character arrays that are allocated
on the stack are used safely.  They are prime targets for buffer overflow
attacks.

Total lines analyzed: 15
Total time 0.000891 seconds
16835 lines per second
```

Saída RATS para fs.c

```
$ rats -d rats-c.xml fs.c

Entries in c database: 310
Analyzing fs.c
fs.c:2: High: fixed size local buffer
Extra care should be taken to ensure that character arrays that are allocated
on the stack are used safely.  They are prime targets for buffer overflow
attacks.

fs.c:5: High: sprintf
Check to be sure that the non-constant format string passed as argument 2 to
this function call does not come from an untrusted source that could have added
formatting characters that the code is not prepared to handle.

fs.c:5: High: sprintf
Check to be sure that the format string passed as argument 2 to this function
call does not come from an untrusted source that could have added formatting
characters that the code is not prepared to handle.  Additionally, the format
string could contain '%s' without precision that could result in a buffer
overflow.

Total lines analyzed: 13
Total time 0.000262 seconds
49618 lines per second
```

Saída PScan para fs.c

```
$ pscan fs.c

fs.c:5 SECURITY: sprintf call should have "%s" as argument 1
```