PROGRAMAÇÃO SEGURA

Rogerio Fernandes Tott

pesquisador – ACME!

Prof. Dr. Adriano Mauro Cansian

coordenador – ACME!

Nós não precisaríamos gastar tanto tempo, dinheiro e esforço em segurança de redes se não tivéssemos uma segurança em software tão ruim

Bruce Schneier

ACME! Laboratório de pesquisa em segurança de redes de computadores

DADOS SOBRE FALHAS EM SOFTWARE

Vulnerabilidades Reportadas - CERT

Vulnerabilities reported

1995-1999

Year	1995	1996	1997	1998	1999
Vulnerabilities	171	345	311	262	417

2000-2005

Year	2000	2001	2002	2003	2004	1Q,2005
Vulnerabilities	1,090	2,437	4,129	3,784	3,780	1,220

Total vulnerabilities reported (1995-1Q,2005): **17,946**

http://www.cert.org/stats/cert_stats.html

ONDE ESTAMOS

Roteiro do Tutorial

Introdução

Segurança e projeto de software

Introdução sobre ataques

Princípios da programação segura

Buffer Overflow

Condições de corrida

Criptografia

Auditoria de software

Conclusões

INTRODUÇÃO SOBRE PROGRAMAÇÃO

Desenvolvimento de Sistemas

Dividido em Fases

>>> Design (Modelagem)

>>> Codificação (“Programação”)

Não começar a construir sua casa sem antes definir onde ficarão os quartos!

INTRODUÇÃO SOBRE PROGRAMAÇÃO

Ferramentas para modelagem

>>> DFD(Diagrama de Fluxo de dados) – Modelagem de sistemas

>>> DER(Diagrama Entidade-Relacionamento) – Banco de dados

>>> UML(Unified Modeling Language) – Orientado a Objetos

>>> Grafos – Mapas, Circuitos, etc...

>>> Autômatos – Compiladores, Protocolos, IA, etc...

>>> Algorítmos!

INTRODUÇÃO SOBRE PROGRAMAÇÃO

Modelagem

conecta
enquanto conectado
    em paralelo
        recebe mensagem
        envia mensagem

Codificação

```
void *rec_pacs()
{
   int n;
   char bf[256];
   while(continuar)
   {
      bzero(bf,256);
      n = recvfrom(sk,bf,256,0,&fr, &ln);
      if (n < 0) error("recvfrom");
      printf(bf);
   }
}
```

INTRODUÇÃO SOBRE PROGRAMAÇÃO

Desenvolvimento de Sistemas Seguros

Programadores bons escrevem sistemas ruins (inseguros)

Por quê?

>>> Livros de programação não citam programação segura!

>>> Professores de programação também não citam!

>>> C/C++ sacrifica segurança em nome da performance!

>>> Existe a mentalidade: Programar = Resolver o Problema!

INTRODUÇÃO SOBRE SEGURANÇA

Quão seguro deve ser um software?

Seguro contra qualquer ataque?

O mais seguro possível?

Depende de quão seguro o software precisa ser!

(e o que é segurança?)

INTRODUÇÃO SOBRE SEGURANÇA

Objetivos em segurança de software

>>> Prevenção
>>> Rastreamento e Auditoria
>>> Monitoramento
>>> Privacidade e Confidencialidade
>>> Segurança em níveis
>>> Anonimato
>>> Autenticação
>>> Integridade

INTRODUÇÃO SOBRE SEGURANÇA

Prevenção

Agir proativamente!

Exemplo:

>>> OpenBSD : 1 buraco em 8 anos! Pensamento proativo em segurança!

INTRODUÇÃO SOBRE SEGURANÇA

Auditoria e Rastreamento

Ataques irão acontecer!

>>> É necessário saber quando aconteceram os ataques. Seus objetivos e suas consequências! (análise forense)

>>> Os mecanismos de auditoria podem sofrer ataques! O que fazer então? (log do log?) >> (log do log do log?) >> (**1**)

INTRODUÇÃO SOBRE SEGURANÇA

Monitoramento

Auditoria em tempo real!

>>> Executada através de software
>>> Executada por seres humanos (“mindware”)

Tem o intuito de detectar e barrar o ataque em tempo real!

INTRODUÇÃO SOBRE SEGURANÇA

Vários níveis de segurança

O que deve ser protegido? e quão protegido?

Podemos ter:

>>> Níveis de privilégio

>>> Níveis de ocultamento (quem pode ver e quem pode modificar?)

Exemplo prático: Fórum Eletrônico

>>> Convidados, usuários, moderadores, administradores, etc...

INTRODUÇÃO SOBRE SEGURANÇA

Anonimato

Uma faca de dois gumes!

>>> Pode proteger contra preconceito e discriminação!
>>> Tem o poder de ocultar crimes!

Oposto em relação ao processo de rastreamento!

O que usar?

INTRODUÇÃO SOBRE SEGURANÇA

Privacidade e confidencialidade*

Os motivos são óbvios!

>>> Proteger segredos!

*um dos pilares da segurança

INTRODUÇÃO SOBRE SEGURANÇA

Autenticação*

Em quem você confia?

Como confiar?

>>> Chaves assimétricas, assinadas digitalmente!

O quanto confiar?

*um dos pilares da segurança

INTRODUÇÃO SOBRE SEGURANÇA

Integridade*

Garantir a veracidade da informação!

Uma atividade dificil de ser executada por meios digitais!

>>> Pode ser usado funções hashing para tentar garantir a integridade!

(mas mesmo isso pode ser burlado)

*um dos pilares da segurança

ONDE ESTAMOS

Roteiro do Tutorial

Introdução

Segurança e projeto de software

Introdução sobre ataques

Princípios da programação segura

Buffer Overflow

Condições de corrida

Criptografia

Auditoria de software

Conclusões

SEGURANÇA & PROJETO DE SOFTWARE

Análise de Risco

Muitos dos requisitos de software vão de encontro com os requisitos de segurança! (e as vezes com os próprios requisitos de software)

SEGURANÇA & PROJETO DE SOFTWARE

Alguns requisitos de software

>>> Usabilidade
>>> Confiabilidade
>>> Eficiência
>>> Viabilidade (Time-to-market)
>>> Simplicidade

SEGURANÇA & PROJETO DE SOFTWARE

Usabilidade

Choque direto com segurança:

Quanto mais seguro, mais dificil de usar!

O usuário final se aborrece ao ter que usar passwords, ao descobrir que sua sessão expirou, etc...

Também vai de encontro com a confiabilidade do sistema!

>>> Todo ser humano tem o direito de errar!

>> Sistemas com tolerância a falhas são chatos de usar

SEGURANÇA & PROJETO DE SOFTWARE

Eficiência

Segurança gera overhead!

>>> Criptografia, Autenticação, Controle de sessão, checagem de limites, análise de expressões...

Também vai de encontro com a simplicidade!

>>> BubbleSort X MergeSort

SEGURANÇA & PROJETO DE SOFTWARE

Viabilidade (Time-to-market)

Gerador de pressão!

>>> Tempo escasso para localização e correção de bugs!
>>> Tempo escasso para projeto de software!

>>> Um time de profissionais experientes tende a conseguir lidar com a situação e desenvolver de forma rápida e segura.

SEGURANÇA & PROJETO DE SOFTWARE

Simplicidade (Kiss)

Pela lógica abaixo, a simplicidade minimiza os bugs!

>>> O número de linhas de código em um sistema é proporcional à complexidade de código desse sistema

>>> O número de bugs do sistema é proporcional ao número de linhas de código desse sistema!

Complexidade gera código que gera bugs!

Vai de encontro com todos os requisitos de segurança.

ONDE ESTAMOS

Roteiro do Tutorial

Introdução

Segurança e projeto de software

Introdução sobre ataques

Princípios da programação segura

Buffer Overflow

Condições de corrida

Criptografia

Auditoria de software

Conclusões

INTRODUÇÃO SOBRE ATAQUES

Como definir ataque a um sistema?

Um ataque em um sistema é qualquer atividade maliciosa intencional contra um sistema. Vejam alguns conceitos importantes: *

>>> Objetivo Final – O objetivo do ataque

>>> Sub-Objetivos – Necessários para se alcançar o objetivo final

>>> Atividades – O ato de executar uma ação maliciosa, com o intuito de alcançar algum dos Sub-Objetivos.

>>> Eventos – O resultado das atividades maliciosas

>>> Consequencias – Os efeitos diretos de um ataque

>>> Impactos – Efeitos indiretos de um ataque

*Segundo *Secure Coding, Principles & Practices*

INTRODUÇÃO SOBRE ATAQUES

Tipos de ataques*

Um ataque pode ser dividido em vários níveis, no que diz respeito ao tipo de vulnerabilidade a ser explorada.

>>> Ataques a nível de projeto – Falhas no algorítmo

>>> Ataques a nível de implementação – Falhas na codificação

>>> Ataques a nível de operação – Falhas na administração do sistema

*Segundo *Secure Coding, Principles & Practices*

INTRODUÇÃO SOBRE ATAQUES

Principais ataques

nível de projeto:*	nível de implementação:*
Ataque *sniffer (Tubo Aspirador**)*	Vazamento de *Buffer*
Homem do Meio	"Porta dos fundos"
Ataque r*eplay*	Falha no tratamento de expressões
Morte de sessão	
Sequestro de sessão	
Condição de corrida	

*Segundo *Secure Coding, Principles & Practices*

**Segundo *Google Language Tools*

INTRODUÇÃO SOBRE ATAQUES

sniffer

O atacante consegue ver os pacotes que passam na rede!

Pode ser defendido com

>>> Uso de switches na camada de enlace (embora não resolva totalmente o problema)

>>> Uso de criptografia com **chaves assimétricas**

Criptografia? Chaves assimétricas?

PARALELO – CRIPTOGRAFIA

Introdução à criptografia

O que é criptografia?

>>> Ato de **embaralhar** as informações com determinada **chave,** de forma que somente o detentor da chave consegue reorganizar a informação.

PARALELO – CRIPTOGRAFIA

Tipos de chaves

Existem dois tipos principais de chaves:

Simétricas

>>> Onde **a mesma chave** é usada para criptografar e decriptografar determinado texto

Assimétricas

>>> Consiste em um par de chaves, **Chave Pública** e **Chave Privada,** onde um texto criptografado com a chave pública de um par de chaves só pode ser decriptografado com a chave privada do mesmo par de chaves. O contrário também é verdadeiro.

PARALELO – CRIPTOGRAFIA

Chaves assimétricas

Chaves públicas:

>>> Todos conhecem

>>> criptografa mensagens destinada ao dono da chave.

>>> verifica autenticidade da mensagem, quando assinada.

Chaves privadas:

>>> Somente o dono conhece

>>> Usada para assinar mensagens

>>> decriptografa as mensagens destinadas ao dono da chave.

INTRODUÇÃO SOBRE ATAQUES

Homem do Meio

O atacante consegue ver os pacotes que passam na rede, além de poder modificar o conteúdo desses pacotes

Pode ser defendido com

>>> Uso de chaves públicas certificadas

PARALELO – CRIPTOGRAFIA

Chaves públicas certificadas

Uma **autoridade certifidadora** atesta que determinada chave pertence a determinada pessoa (física ou jurídica) assinando a chave pública dessa pessoa.

A chave pública da autoridade certificadora necessita ser adquirida de forma **segura**.

INTRODUÇÃO SOBRE ATAQUES

Ataque *Replay*

O atacante grava o processo de autenticação em determinada máquina para depois reproduzi-lo com o intuito de conseguir acesso privilegiado, por exemplo

Pode ser defendido com

>>> Utilização de chaves assimétricas assinadas digitalmente e, sobretudo, utilização de **nonce**.

INTRODUÇÃO SOBRE ATAQUES

Condições de Corrida

O atacante se aproveita de brechas no algorítmo do software para fazer com que o sistema se comporte de forma anômala!

Defesa:

>>> Entender a diferença entre comandos atômicos e não atômicos, e evitar o uso do último destes em partes críticas do sistema.

INTRODUÇÃO SOBRE ATAQUES

Buffer Overflow

O atacante envia uma string malformada que excede o limite do buffer a ela destinado, sobrescrevendo outras variáveis e/ou parte do código, o que pode ocasionar desvio do fluxo de código, podendo também ocorrer a execução de **código malicioso**

Defesa:

>>> Não use C/C++, ou...

>>> Tenha o cuidado de checar os limítes de seus buffers

INTRODUÇÃO SOBRE ATAQUES

"Porta dos fundos"

Ataque realizado na fase de implementação do sistema: O atacante insere código malicioso durante o desenvolvimento do software

Defesa:

>>> Auditoria de código!

INTRODUÇÃO SOBRE ATAQUES

Erro no tratamento de expressões

Devido a falhas no tratamento de expressões, o usuário consegue transpassar as restrições do sistema

Defesa:

>>> Reutilizar código! (dos outros!-)

>>> Tomar muito cuidado no processo de desenvolvimento do algorítmo de tratamento de expressões, dando ênfase no que pode ser aceito em determinada expressão!

ONDE ESTAMOS

Roteiro do Tutorial

Introdução

Segurança e projeto de software

Introdução sobre ataques

Princípios da programação segura

Buffer Overflow

Condições de corrida

Criptografia

Auditoria de software

Conclusões

PROGRAMAÇÃO SEGURA: PRINCÍPIOS

Os 10 mandamentos da construção de *software* seguro

Proteja o elo mais fraco

Pratique defesa em profundidade

Falhe de forma segura

Utilize os princípios do mínimo privilégio

Divida em compartimentos

Simplifique! (KISS)*

Dê prioridade à Privacidade

Lembre que é dificil guardar segredo!

Relute em confiar

Use os recursos públicos

*Keep it simple, stupid

PROGRAMAÇÃO SEGURA: PRINCÍPIOS

Proteja o elo mais fraco

>>> A segurança em um sistema é como uma corrente: Tão forte quanto seu elo mais fraco!

>>> Os atacantes sempre procurarão pelos elos mais fracos!

>>> Uma boa segurança afasta os atacantes ocasionais!

>>> O elo mais fraco pode não ser o sistema, mas sim as pessoas!!!

>>> Criar procedimentos contra engenharia social!

PROGRAMAÇÃO SEGURA: PRINCÍPIOS

Pratique defesa em profundidade

>>> Segurança Redundante. Mais barreiras a serem transpostas

>> O porque de se assaltarem mais postos de gasolina do que banco!

>>> "Divida seus ovos em várias cestas!"

A probabilidade de 2 sistemas falharem ao mesmo tempo é extremamente menor do que a probabilidade de um sistema falhar.

PROGRAMAÇÃO SEGURA: PRINCÍPIOS

Falhe de forma segura!

Sistemas complexos irão falhar!

>>> Analize as possíveis falhas

>>> Atue de forma reativa

>>> Saiba detectar as falhas

>>> Saiba se recuperar das falhas

>>> SAIBA COMO AGIR QUANDO NÃO EXISTE MANEIRA DE SE RECUPERAR DAS FALHAS!

PROGRAMAÇÃO SEGURA: PRINCÍPIOS

Utilize os princípios do mínimo privilégio

Em quem você confia?

>>> Não existe razão para permitir além do necessário

>>> Diminui a gravidade das consequências no caso de comprometimento do sistema

>>> Quando bem implementado, dificulta a escalada de privilégios.

PROGRAMAÇÃO SEGURA: PRINCÍPIOS

Divida em compartimentos!

Não exponha seu sistema!

>>> Dividir seu sistema em módulos, para que o comprometimento de um dos módulos não comprometa o funcionamento dos outros módulos

>>> Siga o exemplo da segurança presente em cadeias e submarinos!

PROGRAMAÇÃO SEGURA: PRINCÍPIOS

Simplifique

Pra que complicar?

>>> Não reinvente a roda! Utilize o que já existe!

>>> Complexidade atrai bugs!

>>> Deixe a segurança como padrão!

>> Usuário Final não lê manual!

*** Este princípio vai de encontro com a segurança em profundidade e com o uso de compartimentalização! “cuidado com o excesso de cestas de ovos!”

PROGRAMAÇÃO SEGURA: PRINCÍPIOS

Dê prioridade à privacidade

Privacidade do usuário:

*** Vai de encontro com a usabilidade do sistema.

>>> Proteja as informações pessoais dos clientes de seu software.

>>> Guarde o mínimo possível de informações confidenciais.

Privacidade do sistema:

>>> Não mostre informações desnecessárias.

>>> Minta sobre seu sistema!

>> engane o Black-Hat!

PROGRAMAÇÃO SEGURA: PRINCÍPIOS

Lembre-se que é dificil guardar segredos!

>>> Código Binário não é indecifrável!

>> Não guarde chaves criptográficas simétricas em seu binário.

>> Seja relutante em confiar na segurança por obscuridade.

PROGRAMAÇÃO SEGURA: PRINCÍPIOS

Relute em confiar!

>>> O cliente não deve confiar no servidor em um sistema em rede.

>>> A reciprocra é verdadeira.

>>> Confie o mínimo possível

> em seu sistema operacional

> em código de terceiros

> em arquivos de sistemas

> em TUDO!

PROGRAMAÇÃO SEGURA: PRINCÍPIOS

USE os recursos já disponíveis!

>>> Evite reinventar a roda.

>>> O que a comunidade analizou provavelmente tem menos erros do que aquilo que apenas você analizou!

>>> Seja consciente! (não use qualquer coisa)

ONDE ESTAMOS

Roteiro do Tutorial

Introdução

Segurança e projeto de software

Introdução sobre ataques

Princípios da programação segura

Buffer Overflow

Condições de corrida

Criptografia

Auditoria de software

Conclusões

BUFFER OVERFLOW

Introdução

>>> A maioria absoluta das falhas de segurança encontradas em softwares dizem respeito a estouro de buffer! (+ de 50%)

>>> O que é um *buffer overflow*?

>> É quando colocamos mais dados em um buffer do que este buffer está preparado para receber.

>>> Quais são as consequencias?

>> Várias, desde alteração de valores em variáveis até execução de **código malicioso**!

BUFFER OVERFLOW

Porque ocorre?

>>> Falta de checagem de limites de buffer

>>> Checagem incorreta de limites!

>>> USO DE FUNÇÕES INSEGURAS

>> exemplo mais famoso: gets();

BUFFER OVERFLOW

Como se prevenir

>>> Não utilize funções inseguras.

>>> Saiba utilizar as funções seguras da maneira certa.

>>> Verifique os erros!

BUFFER OVERFLOW

Funções de alto risco

Função de risco	Solução
gets(char *s)	utilizar fgets(char *s, int size, stdin)
strcpy(dest, src) strcat(dest, src)	Utilizar strncpy e strncat
sprintf	use **snprintf**, ou idem abaixo

BUFFER OVERFLOW

Funções de alto risco

Função de risco	Solução
scanf(), sscanf(), fscanf(), vfscanf(), vsprintf(), vscanf(), vsscanf()	Use especificadores de precisão (ex: "%**4**s")
strecpy(output, input, exceptions); streadd(output, input, exceptions);	tenha a certeza de ter alocado **4 vezes** o tamanho da entrada para a saida

BUFFER OVERFLOW

Funções de risco considerável

Função de risco	Solução
strtrns(string, old, new, result)	calcule o tamanho máximo de result, usando:**(strlen (string)/strlen(old))*strlen(new)).*** Ou calcular manualmente o tamanho de result

*usada somente se strlen(old) < strlen(new)

BUFFER OVERFLOW

Funções de risco considerável

Função de risco	Solução
realpath(path, resolved_path)*	o buffer de resolved_path **precisa ser** do tamanho de MAXPATHLEN. path **não pode** ser maior que MAXPATHLEN
syslog()*, getopt()*, getopt_long()*, getpass()*	Trunque todas as strings de entrada para um tamanho rasoável antes de passá-las para essas funções

*Podem ser mais seguras, dependendo da versão utilizada

BUFFER OVERFLOW

Funções de risco médio e baixo

Função de risco	Solução
getchar(), fgetc(), getc(), read()	Lembre-se de checar os limites de seu buffer caso estiver usando estas funções em um looping
realpath(path, resolved_path)*	Tenha certeza de que seu buffer é tão grande quanto você diz que é!

BUFFER OVERFLOW

Tipos de *Buffer Overflow*

2 tipos básicos:

>>> *Stack Overflow*

>> Também conhecido como esmagamento de pilha.

>> MUITO GRAVE!!!

>>> *Heap Overflow*

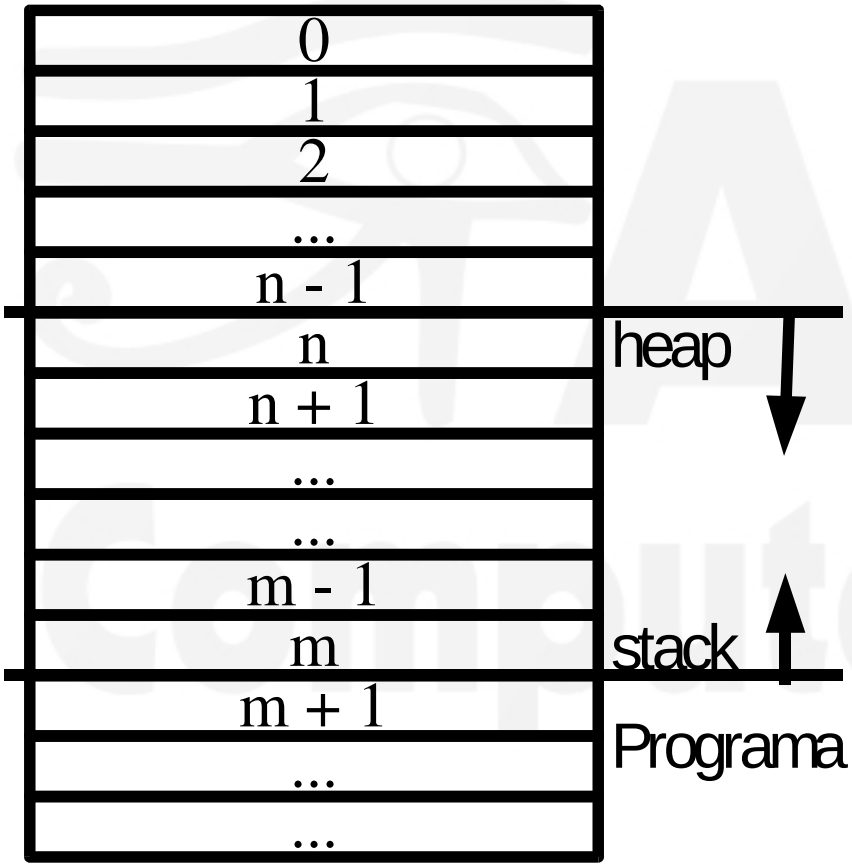
>> Esmagamento de "monte"!

>> GRAVE!

Pilha? Monte?

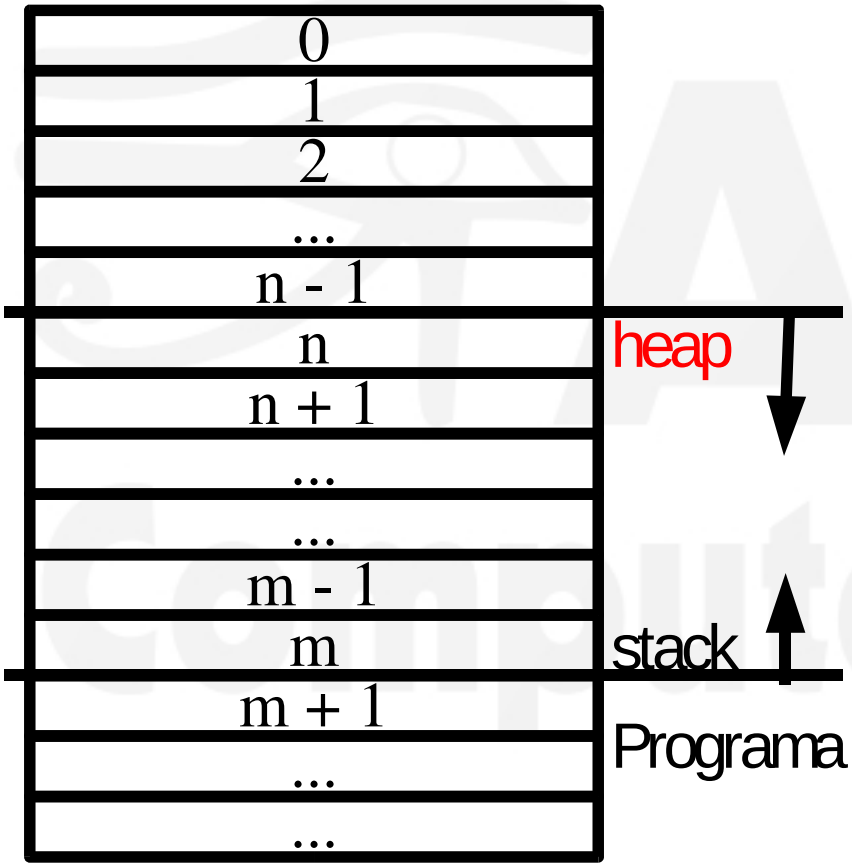
PARALELO - PROGRAMAÇÃO

Estrutura de um programa



PARALELO - PROGRAMAÇÃO

Heap



>>> Armazena dados alocados dinamicamente durante a execução do programa(**Malloc()**)

>>> Cresce em direção à pilha (stack)

HEAP OVERFLOW

introdução

>>> Funciona sobre as variáveis dinâmicas do programa.

>>> Dificil de ser explorado

>> É necessário descobrir quai variáveis tem ligação com a segurança do programa

>> Dificuldade aumenta quando não se tem acesso ao código fonte

HEAP OVERFLOW

exemplo conceitual – versão 0

programa

```
int main(int *argc, char *argv[])
{
   char *nome;
   char *prioridade;
   nome = (char*)malloc(10*sizeof(char));
   prioridade=(char*)malloc(2*sizeof(char));
   strcpy(prioridade,argv[1]);
   strcpy(nome,argv[2]);
   printf("prioridade = %s\n",prioridade);
}
```

saída

```
./ex0 5 12345678900
prioridade = 5
```

HEAP OVERFLOW

exemplo conceitual – versão 1

programa

```
int main(int *argc, char *argv[])
{
   char *nome;
   char *prioridade;
   nome = (char*)malloc(10*sizeof(char));
   prioridade = (char*)malloc(2*sizeof(char));
   printf("%p = nome\n",nome);
   printf("%p = prioridade\n",prioridade);
}
```

saída

```
/ex0 5 1234567890
0x80496a8 = nome
0x80496b8 = prioridade
```

HEAP OVERFLOW

exemplo conceitual – versão 2

programa

```
int main(int *argc, char *argv[]) {
  char *nome;  char *prioridade;
  nome = (char*)malloc(10*sizeof(char));
  prioridade = (char*)malloc(2*sizeof(char));
  strcpy(nota,argv[1]);
  strcpy(nome,argv[2]);
  while (nome < (prioridade + 4))  {
    printf("%p = nome -> %c, valor = %x\n",
nome,isprint(*nome)?*nome:'?',(unsigned int)*nome);
    nome++;  }
}
```

saída

```
./ex01 5 1234567890
0x8049788 = nome -> 1, valor = 31
...
0x8049791 = nome -> 0, valor = 30
0x8049792 = nome -> ?, valor = 0
0x8049793 = nome -> ?, valor = 0
0x8049794 = nome -> ?, valor = 11
0x8049795 = nome -> ?, valor = 0
0x8049796 = nome -> ?, valor = 0
0x8049797 = nome -> ?, valor = 0
0x8049798 = nome -> 5, valor = 35
0x8049799 = nome -> ?, valor = 0
0x804979a = nome -> ?, valor = 0
0x804979b = nome -> ?, valor = 0
```

HEAP OVERFLOW

exemplo conceitual – versão final

programa

```
int main(int *argc, char *argv[])
{
  char *nome;
  char *nota;
  nome = (char*)malloc(10*sizeof(char));
  nota = (char*)malloc(2*sizeof(char));
  strcpy(nota,argv[1]);
  strcpy(nome,argv[2]);
  printf("prioridade = %s\n",prioridade);
}
```

saída

```
./ex0 5 1234567890.....0
prioridade = 0
```

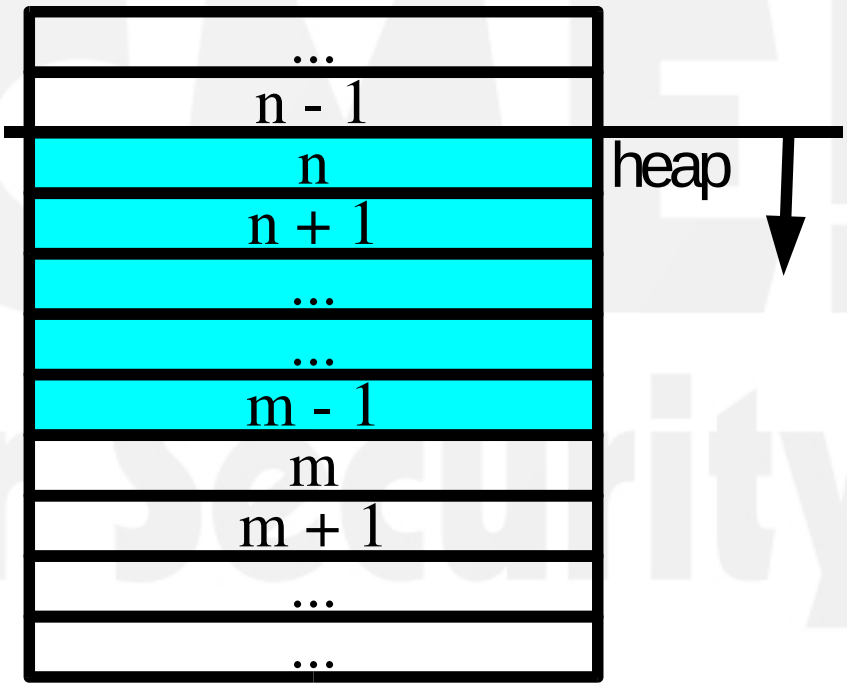
HEAP OVERFLOW

exemplo visual – parte 1

programa

```
int main(int *argc, char *argv[])
{
  char *nome;
  char *nota;
  nome = (char*)malloc(10*sizeof(char));
  prioridade = (char*)malloc(2*sizeof(char));
  strcpy(prioridade,argv[1]);
  strcpy(nome,argv[2]);
  printf("prioridade = %s\n",prioridade);
}
```

memória



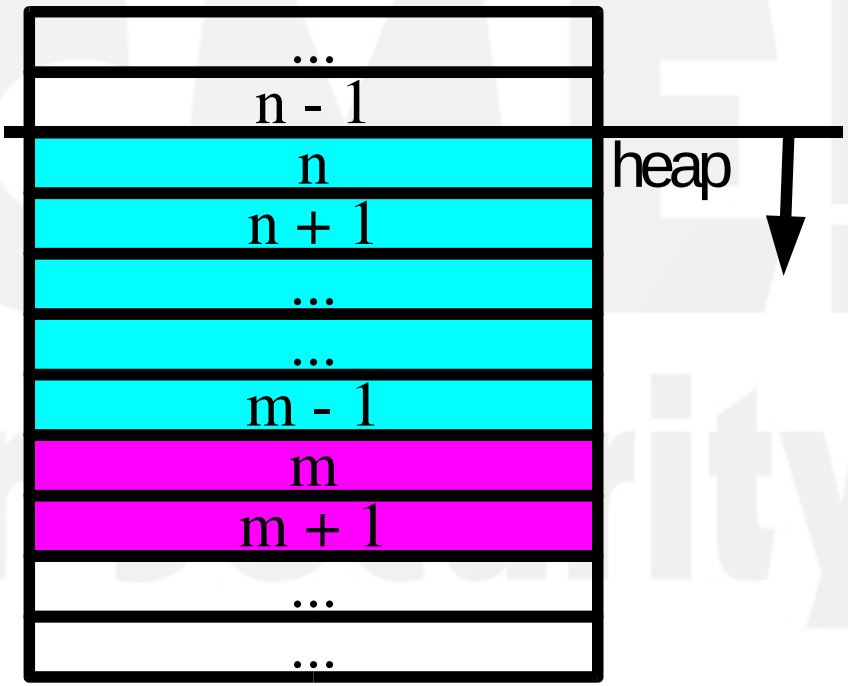
HEAP OVERFLOW

exemplo visual – parte 2

programa

```
int main(int *argc, char *argv[])
{
  char *nome;
  char *nota;
  nome = (char*)malloc(10*sizeof(char));
  prioridade = (char*)malloc(2*sizeof(char));
  strcpy(prioridade,argv[1]);
  strcpy(nome,argv[2]);
  printf("prioridade = %s\n",prioridade);
}
```

memória



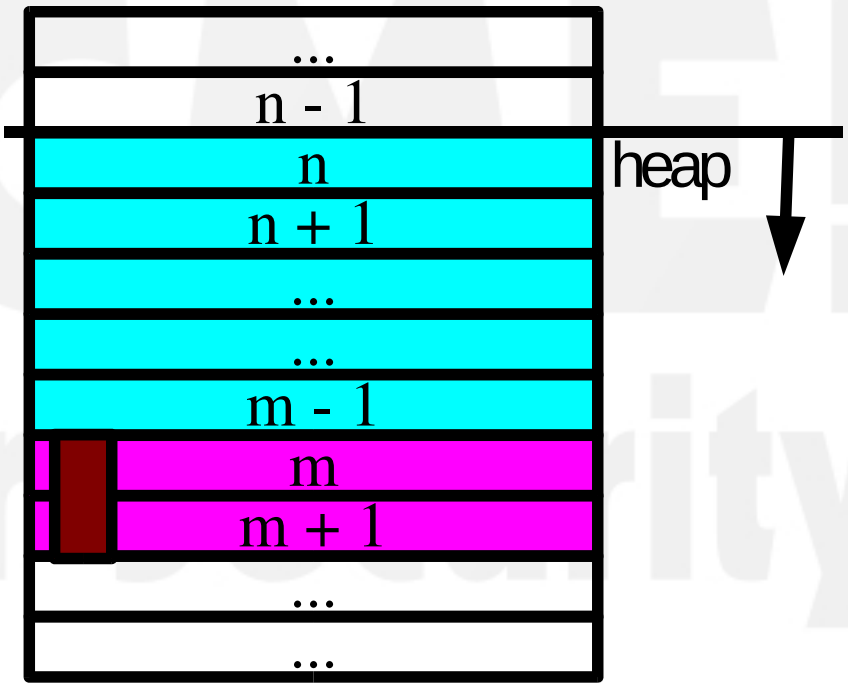
HEAP OVERFLOW

exemplo visual – parte 3

programa

```
int main(int *argc, char *argv[])
{
  char *nome;
  char *nota;
  nome = (char*)malloc(10*sizeof(char));
  prioridade = (char*)malloc(2*sizeof(char));
  strcpy(prioridade,argv[1]);
  strcpy(nome,argv[2]);
  printf("prioridade = %s\n",prioridade);
}
```

memória



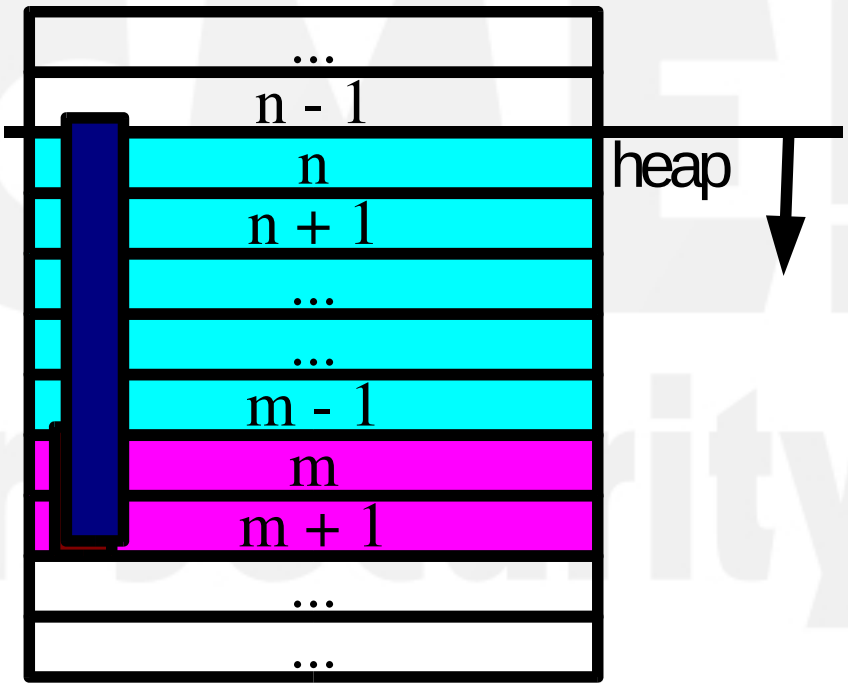
HEAP OVERFLOW

exemplo visual – parte 3

programa

```
int main(int *argc, char *argv[])
{
  char *nome;
  char *nota;
  nome = (char*)malloc(10*sizeof(char));
  prioridade = (char*)malloc(2*sizeof(char));
  strcpy(prioridade,argv[1]);
  strcpy(nome,argv[2]);
  printf("prioridade = %s\n",prioridade);
}
```

memória



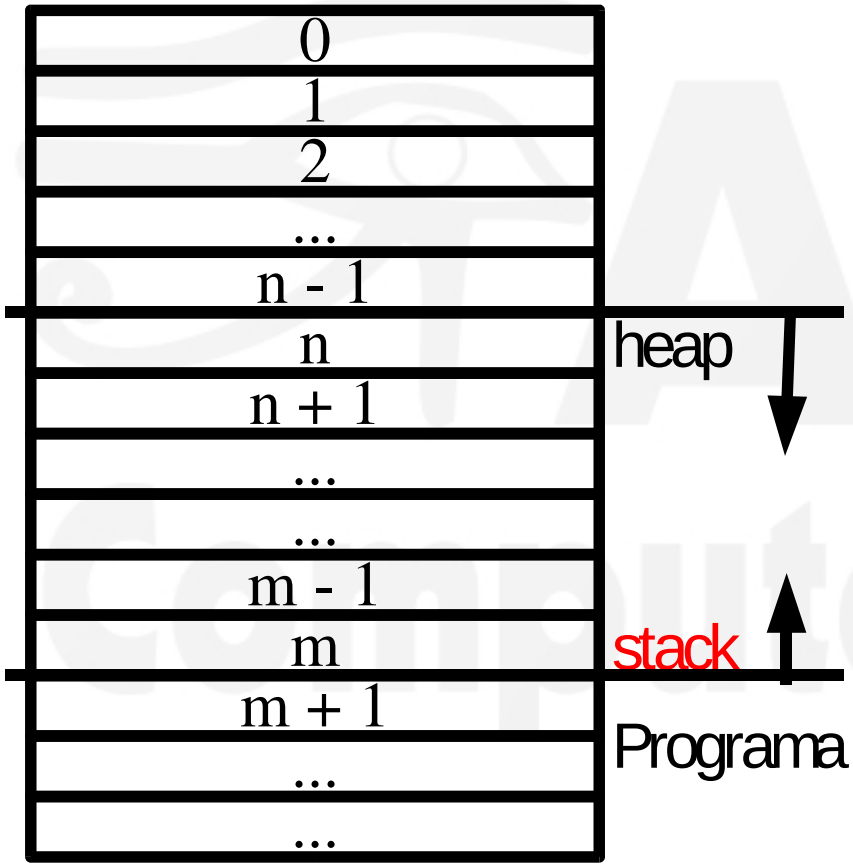
STACK OVERFLOW

introdução

>>> O que vem a ser a pilha?

PARALELO - PROGRAMAÇÃO

Stack



>>> Armazena variáveis locais, parâmetros de funções e **program counter** quando ocorre a chamada de funçoes

>>> Cresce em direção ao heap! (decresce)

ASSEMBLY & ARQUITETURA

Estrutura do processador

Composto basicamente pela ULA(unidade lógica e aritmética), UC(unidade de controle), e e Registradores

>>> ULA: Local onde são efetuadas as operações matemáticas.

>>> UC: Responsavel pela leitura de instruções da memória.

' >>> Registradores: Repositório de informações (identico a variáveis!)

ASSEMBLY & ARQUITETURA

Aguns registradores importantes

Dentre os vários registradores presentes na cpu, temos 2 cujo funcionamento merece destaque:

>>> PC (*program counter* – contador de programa):

>> Armazena o endereço em memória da próxima instrução a ser executada

>>> SP(stack pointer – ponteiro da pilha):

>> Armazena o endereço em memória do "topo" da pilha.

Ambos são PONTEIROS

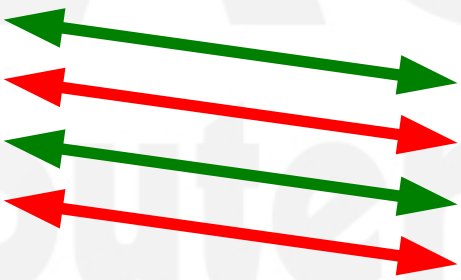
PARALELO - PONTEIROS

variáveis que armazenam **endereço de memória**

Em uma alusãoa vetores, seria a variável que armazena o índice de um vetor. Veja exemplo

```
int main(){
   int *endereco;

   endereco = 10;
   *endereco = 5;
   endereco++;
   *endereco = 7;
}
```



```
int main(){
   int indice;
   int vetor[50];

   indice = 10;
   vetor[indice] = 5;
   indice++;
   vetor[indice] = 7;
}
```

Para fins didáticos! O correto seria usar: `endereco = malloc(50*sizeof(int));`

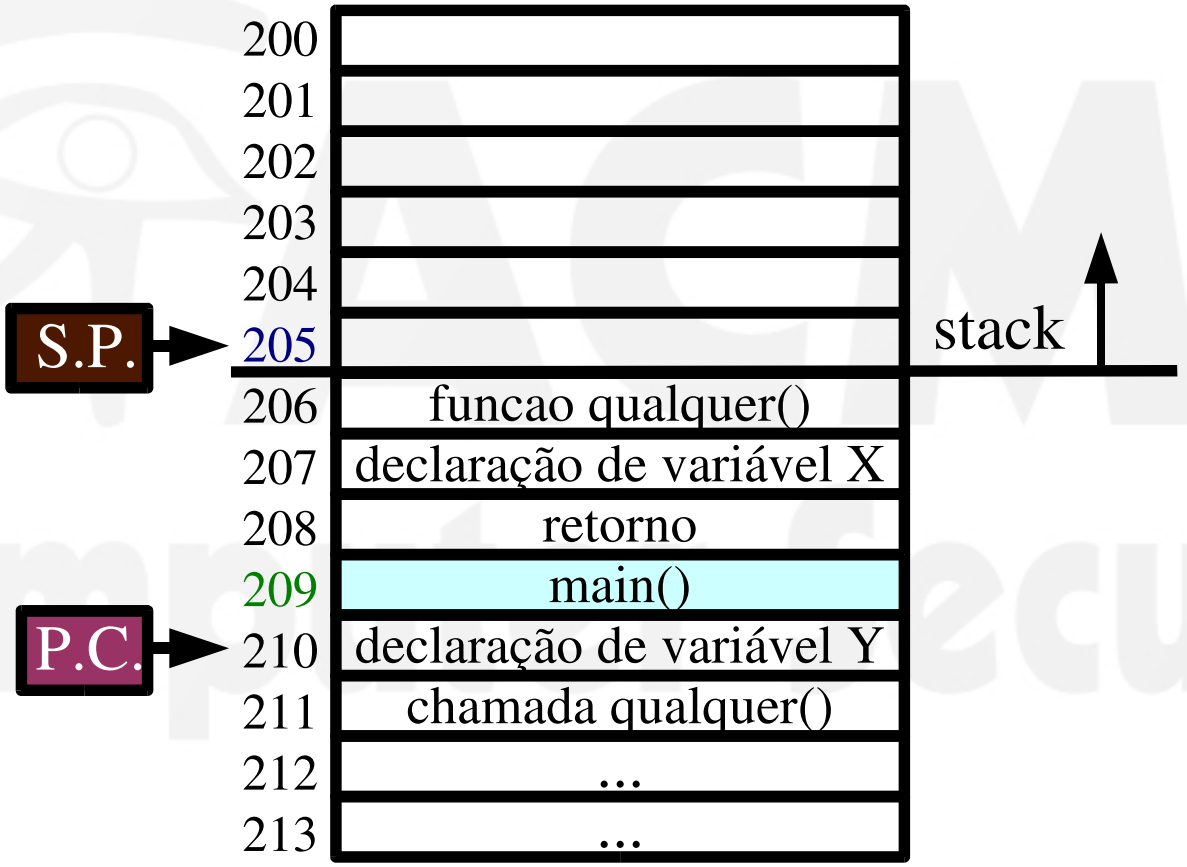
PARALELO - PONTEIROS

Exemplo prático

```
int main(){                                  int main(){
   int *endereco;                               int indice;
   endereco = malloc(50*sizeof(int));           int vetor[50];
                                                indice = 0;
   *(endereco + 10) = 5;                        vetor[10] = 5;
   endereco++;                                  indice++;
   *endereco = 7;                               vetor[indice] = 7;
   *(endereco + 10) = 9                         vetor[11] = 9;
}                                            }
```

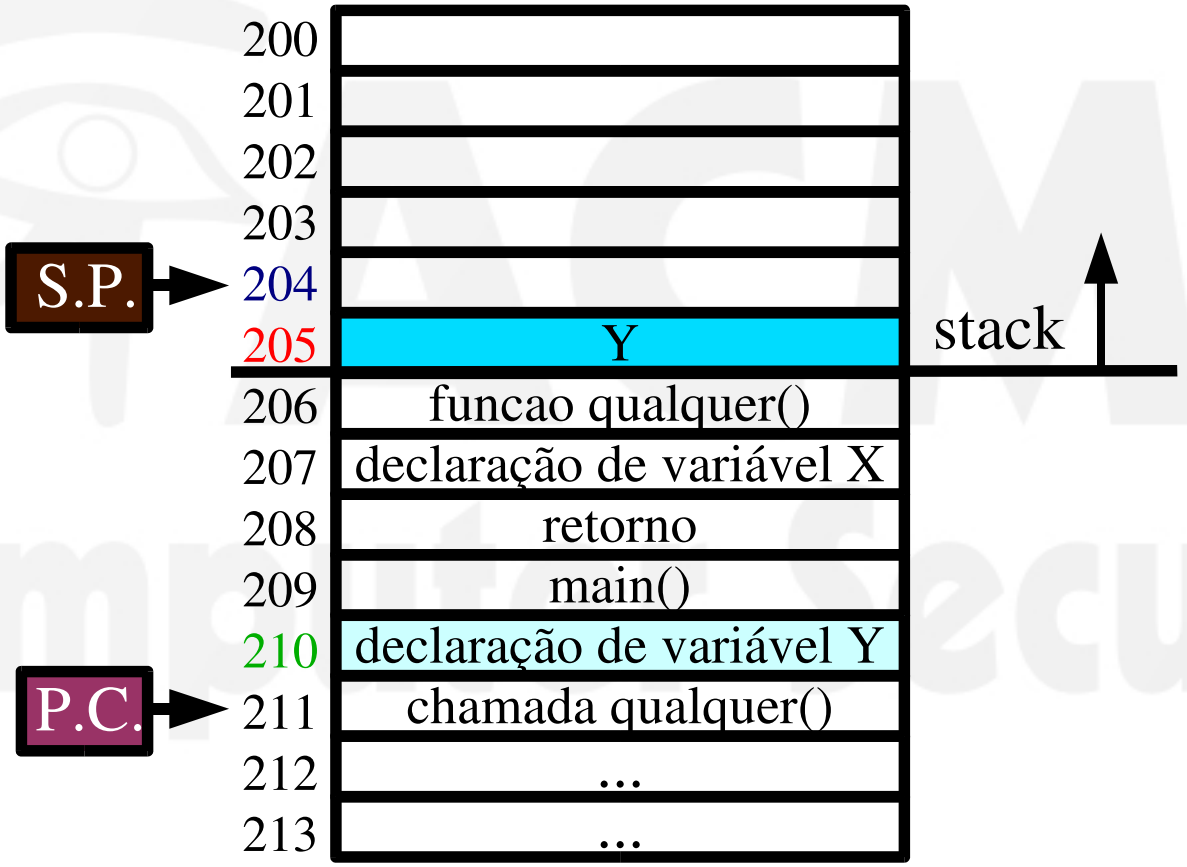
ASSEMBLY & ARQUITETURA

Chamada de funções



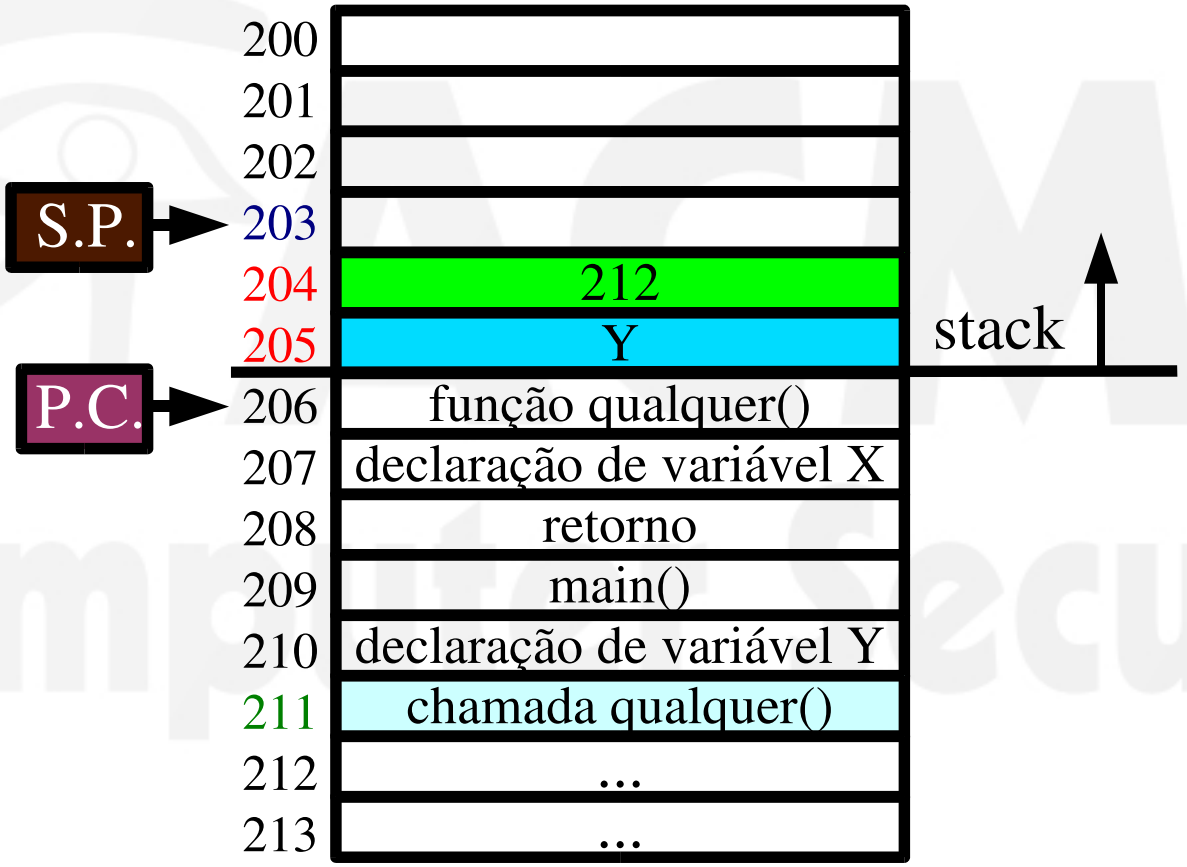
ASSEMBLY & ARQUITETURA

Chamada de funções



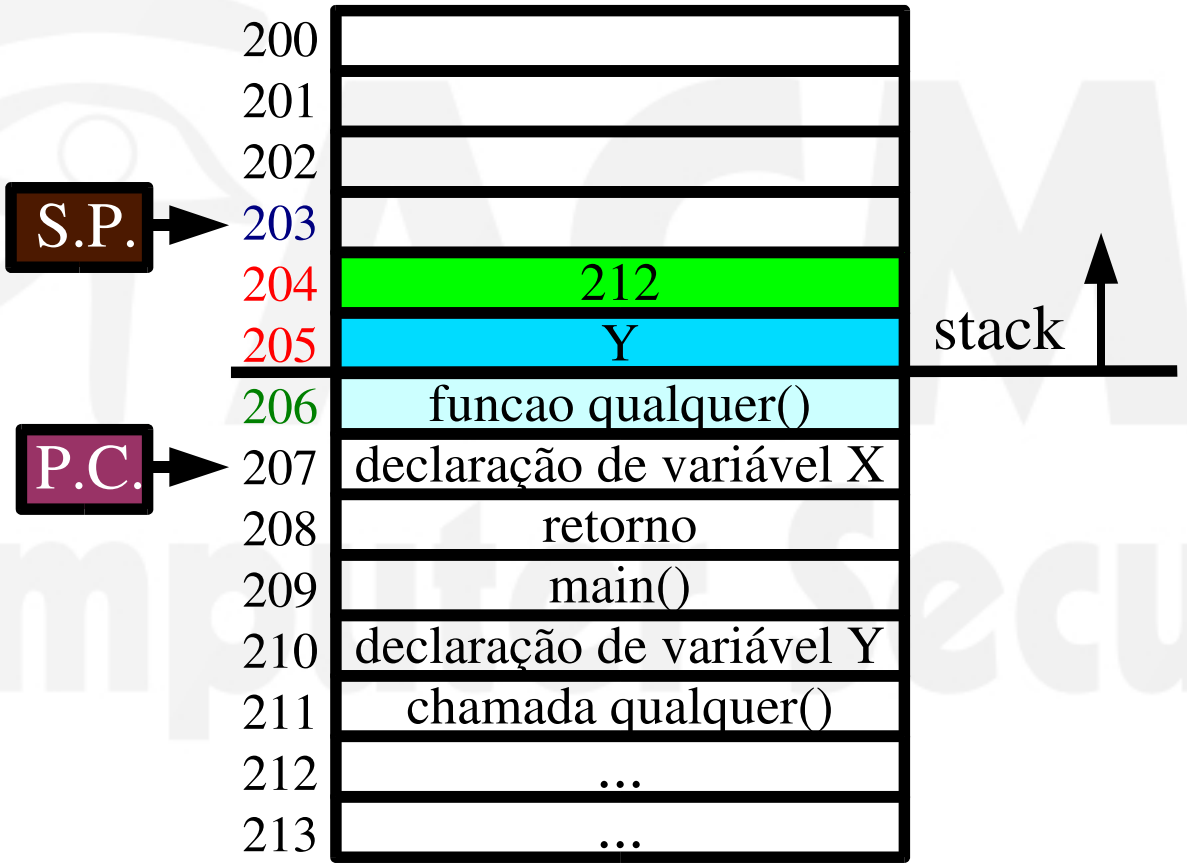
ASSEMBLY & ARQUITETURA

Chamada de funções



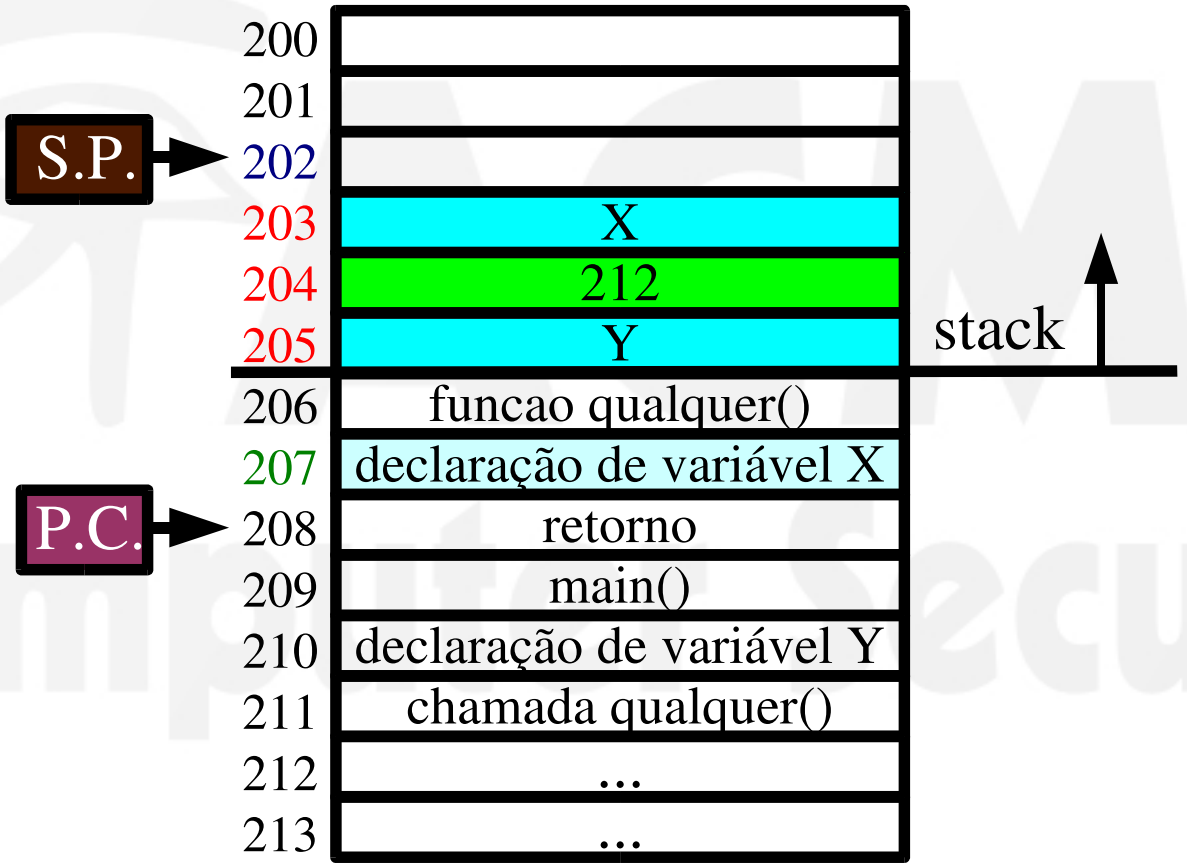
ASSEMBLY & ARQUITETURA

Chamada de funções



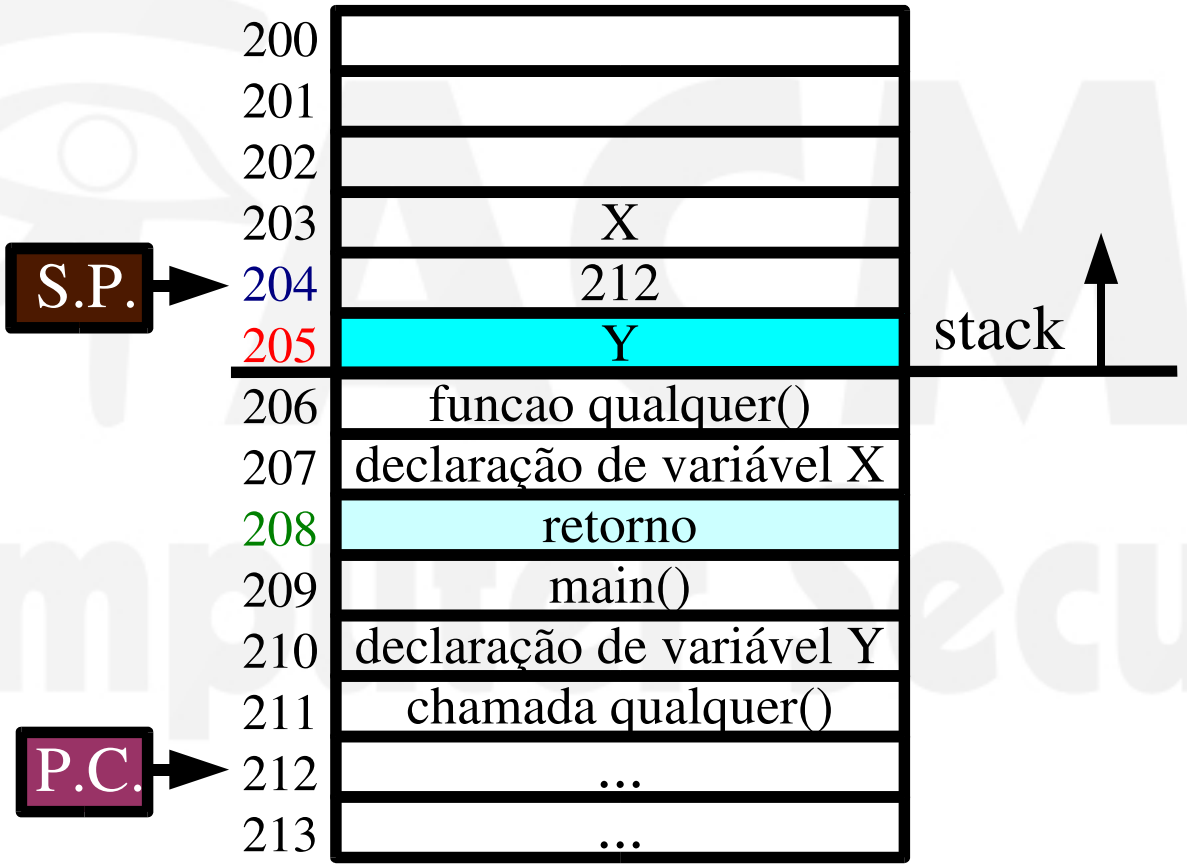
ASSEMBLY & ARQUITETURA

Chamada de funções



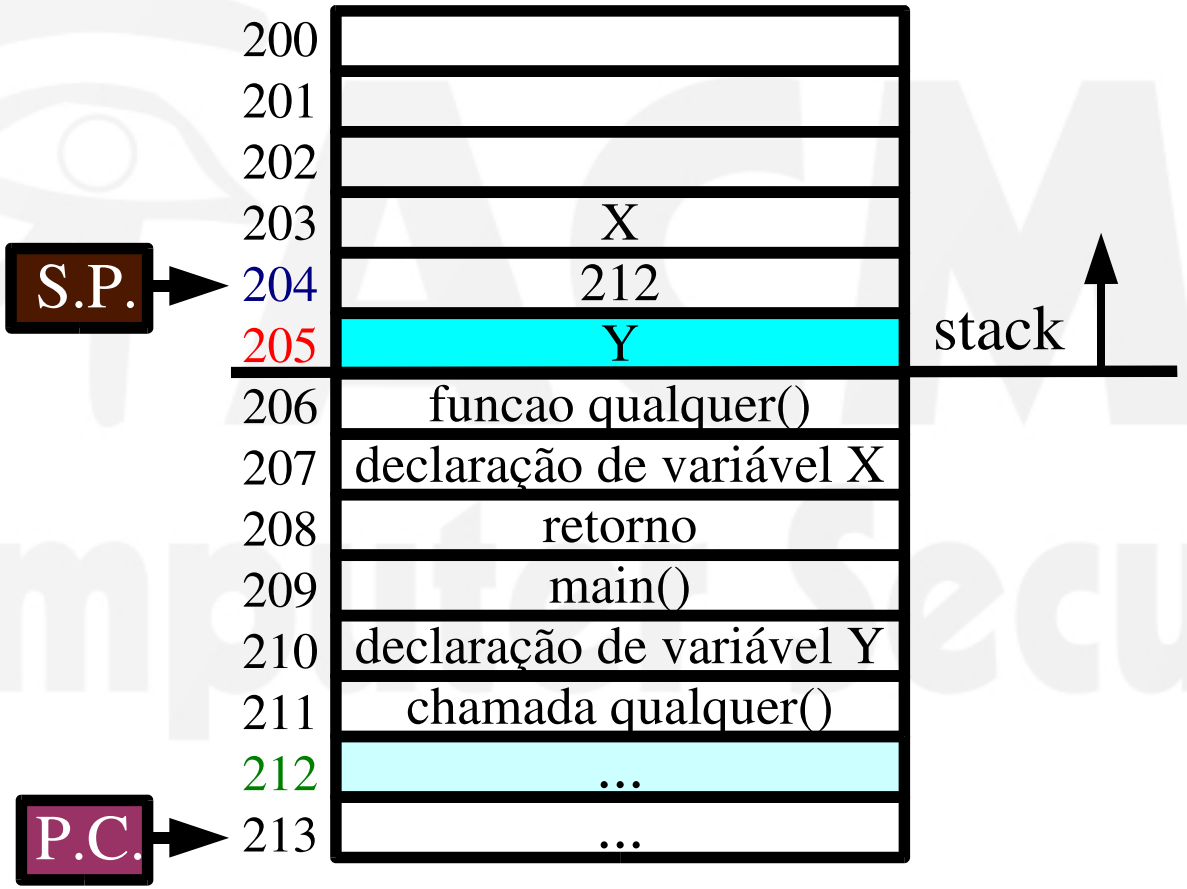
ASSEMBLY & ARQUITETURA

Chamada de funções



ASSEMBLY & ARQUITETURA

Chamada de funções



STACK OVERFLOW

introdução

>>> Sobreposição de dados que se encontram na pilha

>>> Principais efeitos

>>> Modificação de dados

>>> Alteração de fluxo do programa

>>> Execução de código arbitrário!

Ao alterar o valor de retorno da função, podemos alterar arbitrariamente o fluxo do programa. Com isso, podemos executar CODIGO MALICIOSO!

STACK OVERFLOW

funcionamento

>>> Principal Objetivo: Execução de código malicioso (shell)

>>> inserção de shellcode em algum buffer.

>>> Alteração do endereço de retorno da função

>>> Execução de código arbitrário!

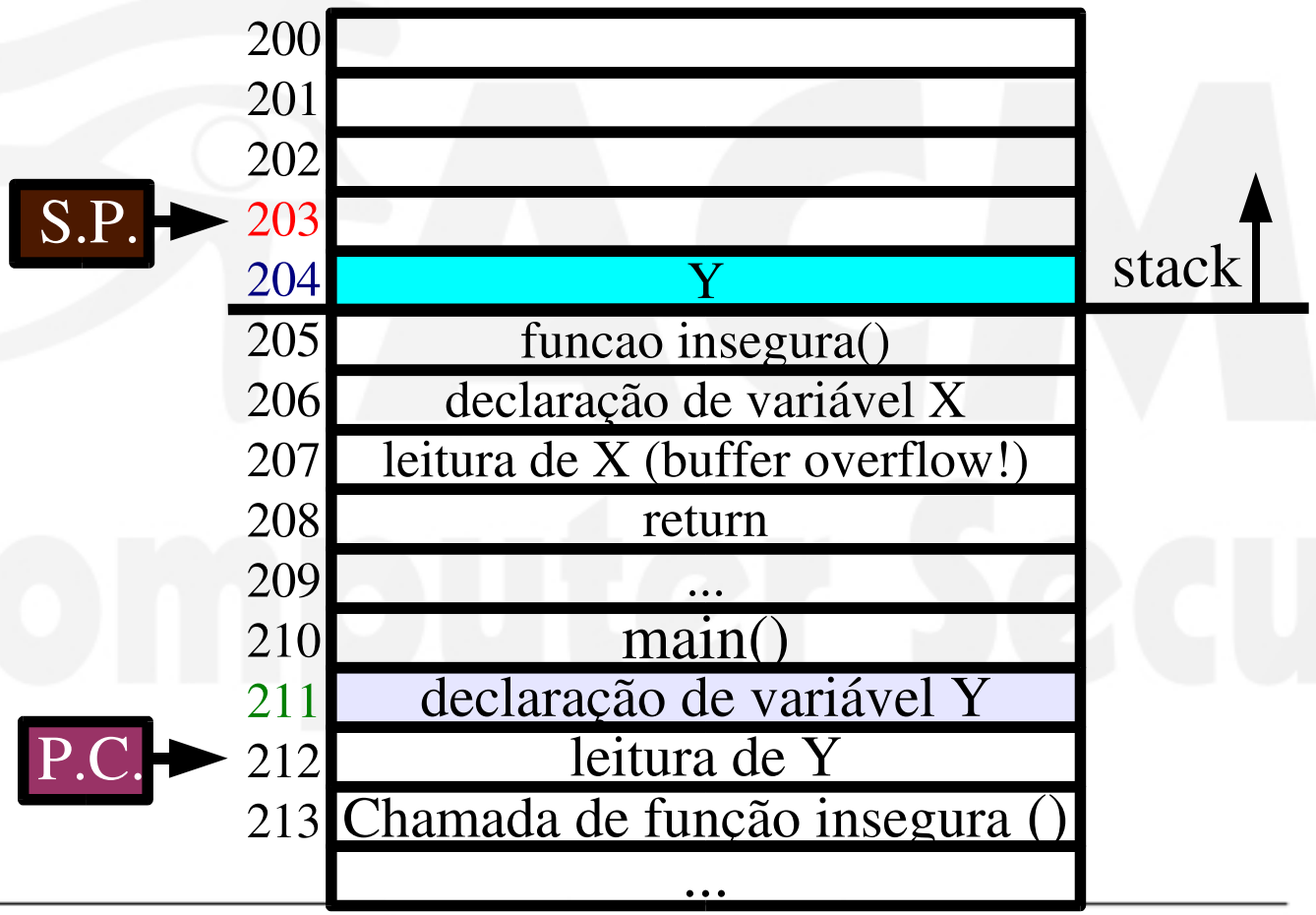
Vale lembrar: main() também é uma função, chamada pelas bibliotecas internas do C/C++.

STACK OVERFLOW

Exemplo Visual

STACK OVERFLOW

Exemplo Visual



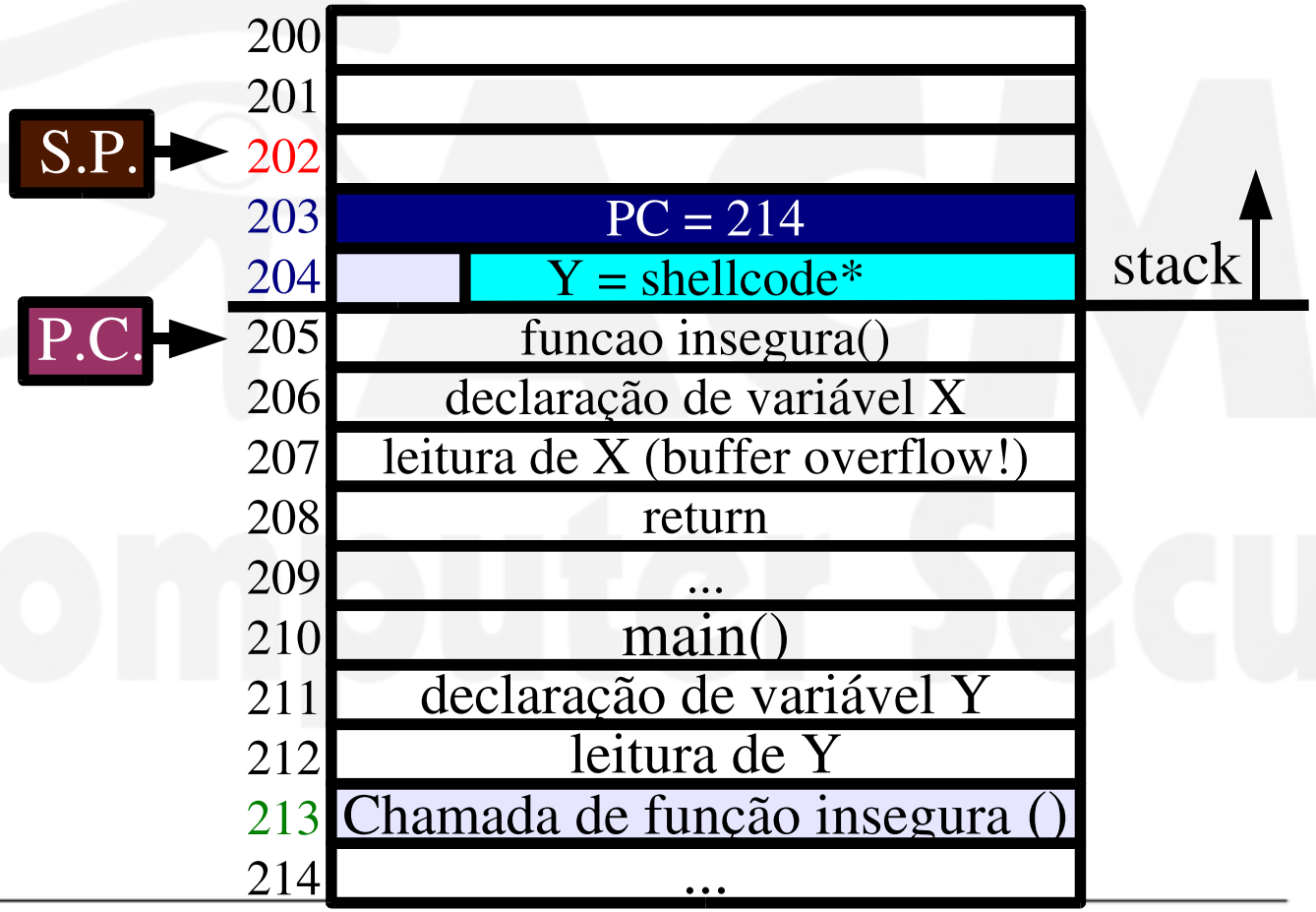
STACK OVERFLOW

Exemplo Visual – inserção de shellcode*

***shellcode** = código binário cujo objetivo consiste em executar um shell(ex: bash)

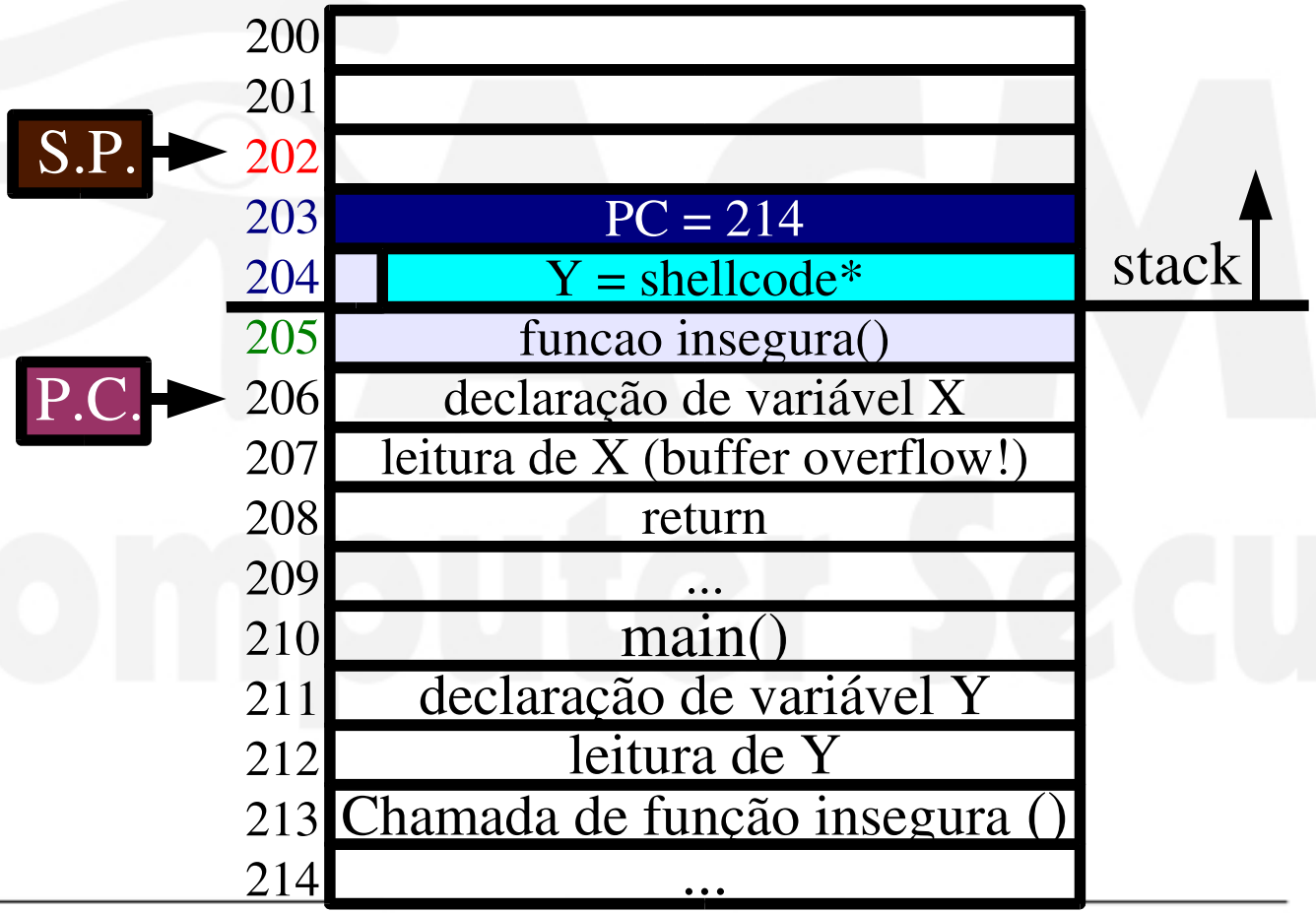
STACK OVERFLOW

Chamada de função – PC inserido na pilha



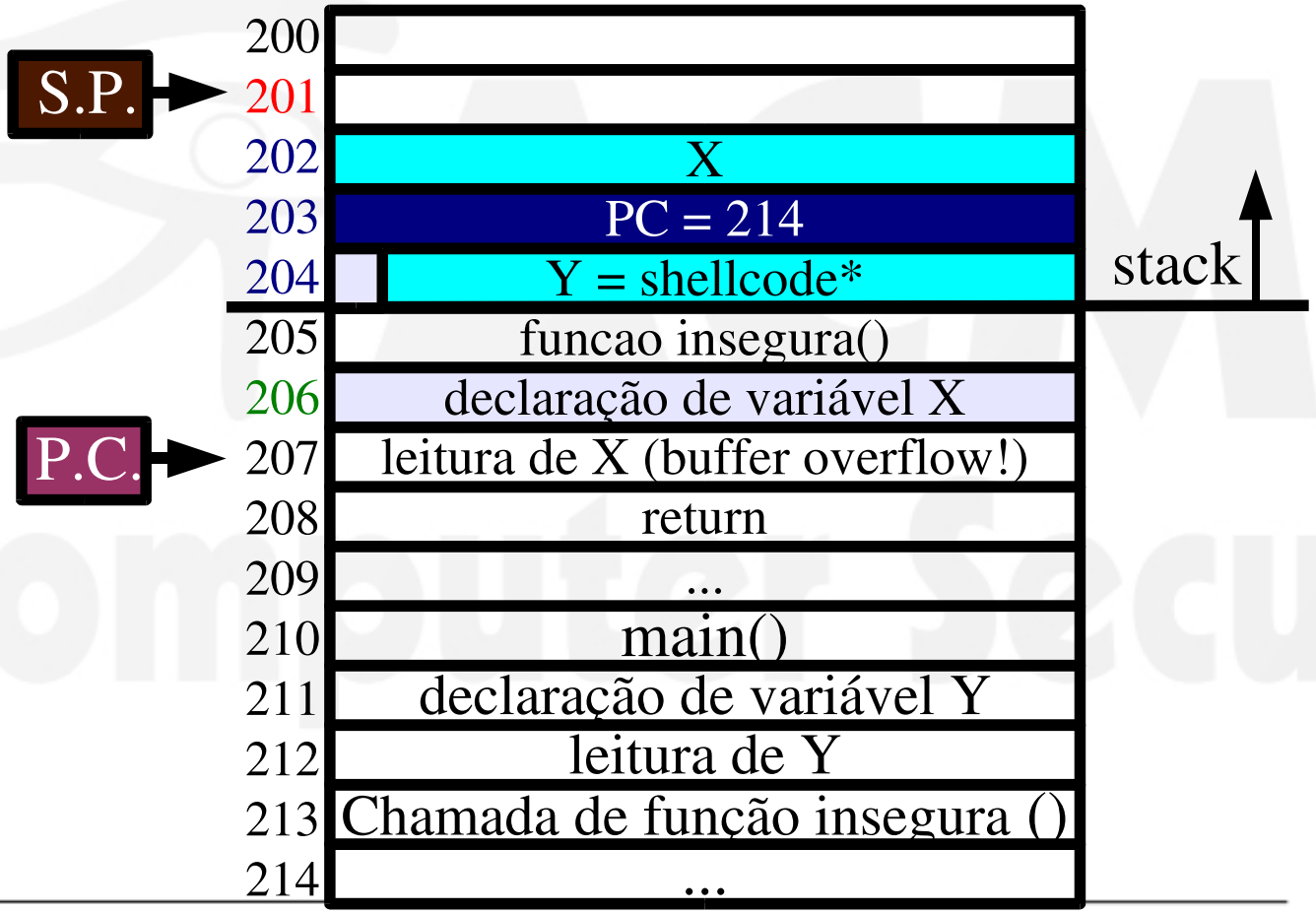
STACK OVERFLOW

Chamada de função – Início da função



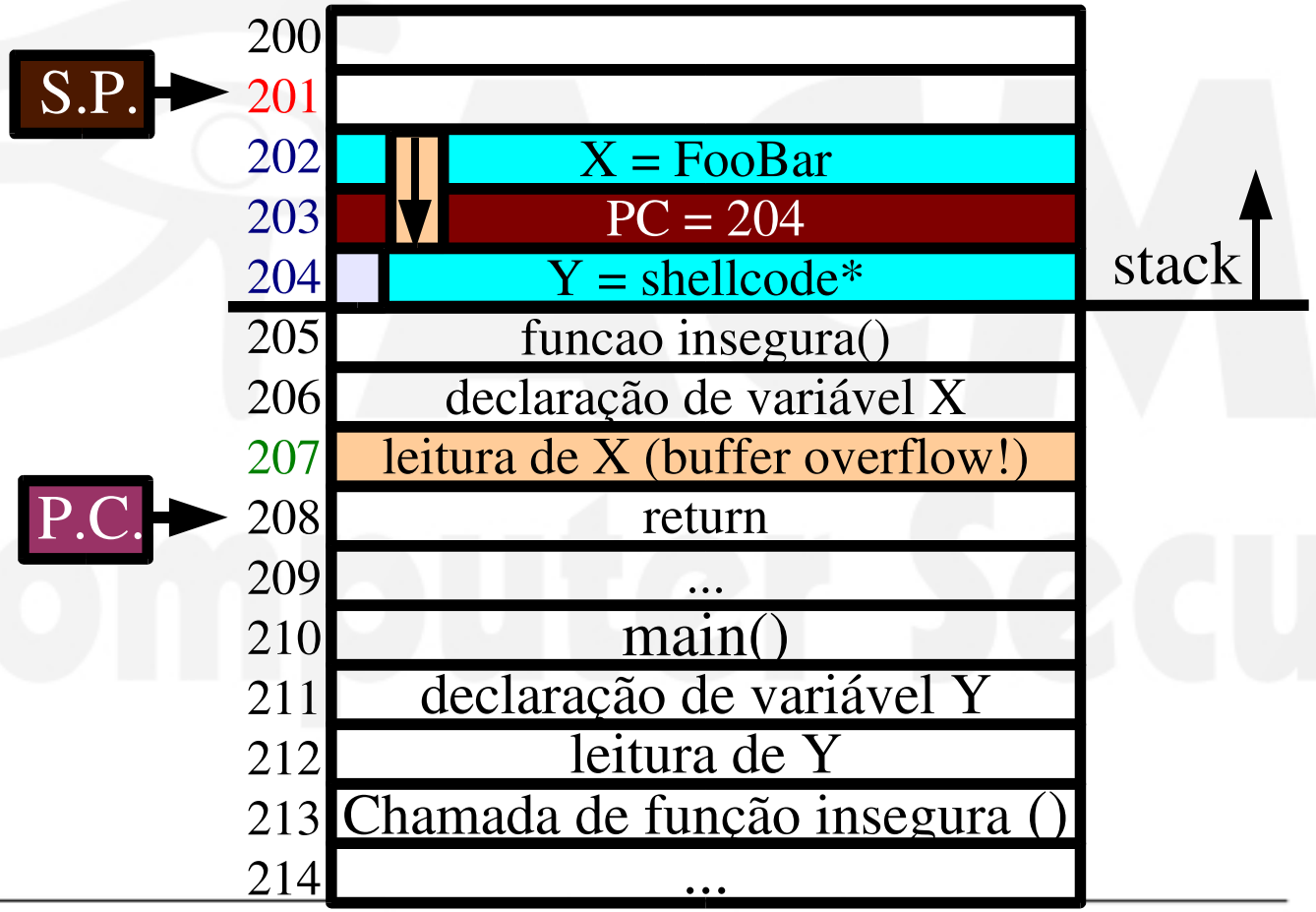
STACK OVERFLOW

Declaração de buffer a ser explorado



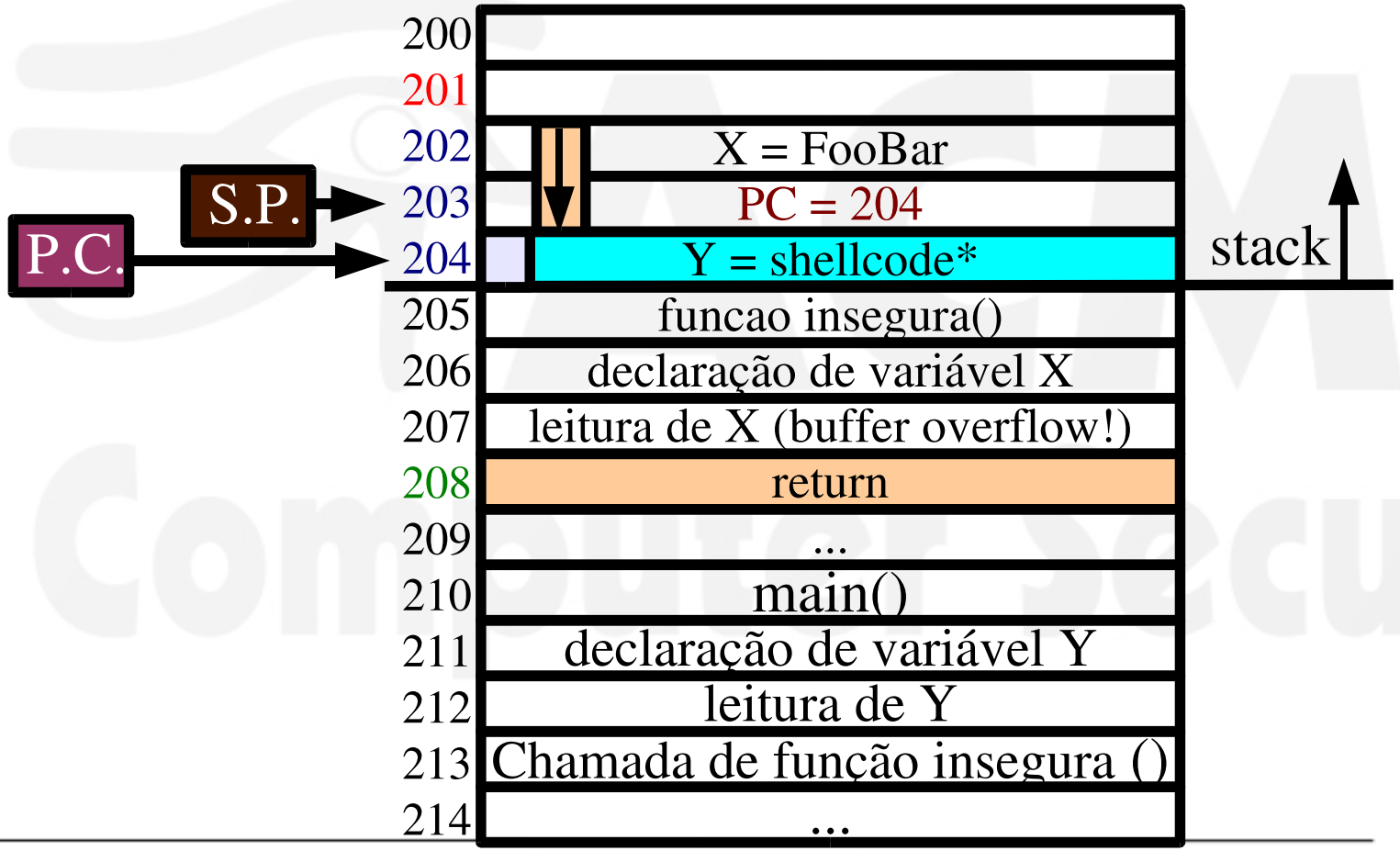
STACK OVERFLOW

BUFFER OVERFLOW



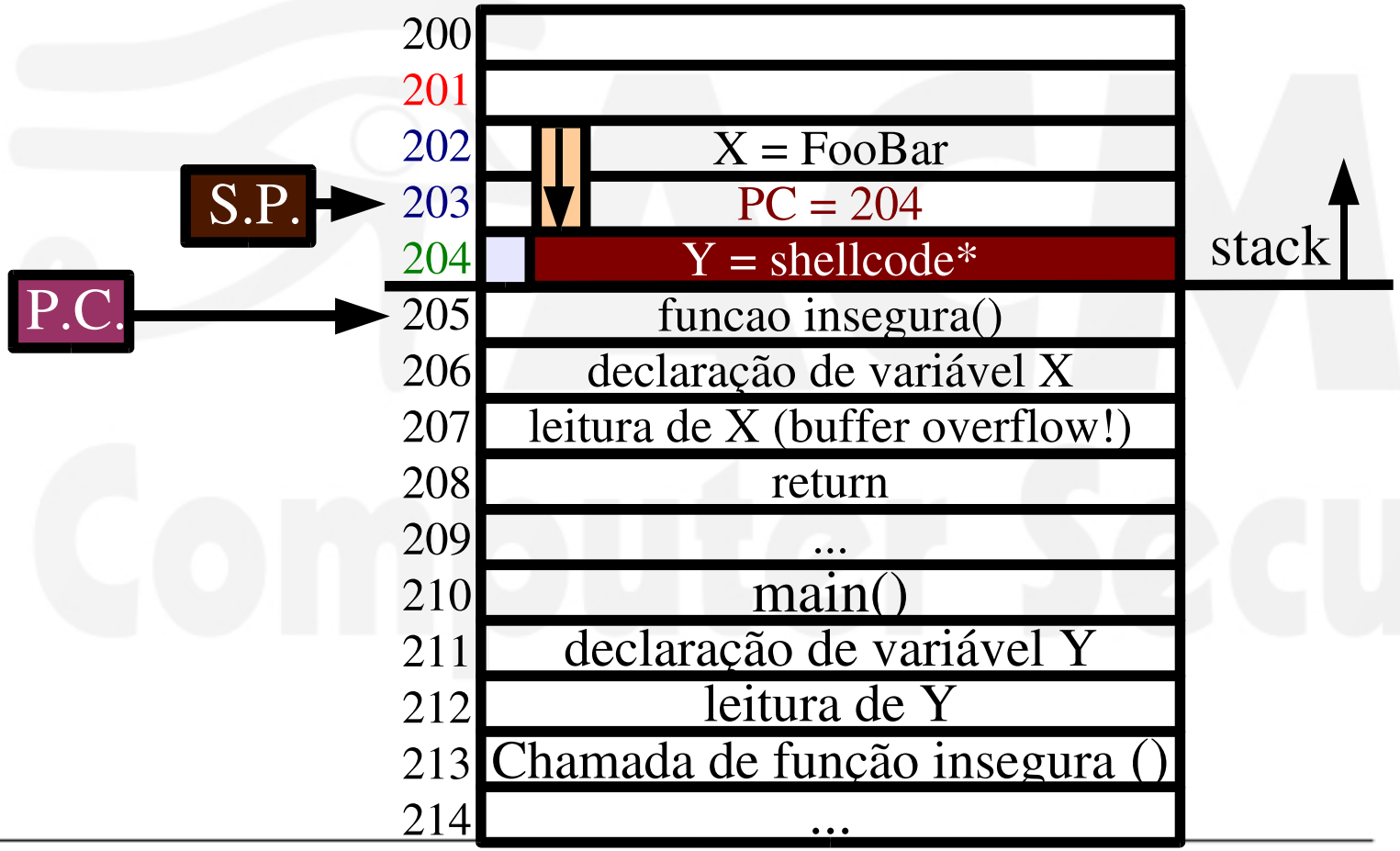
STACK OVERFLOW

Retorno da função insegura



STACK OVERFLOW

Execução de Shellcode!



BUFFER OVERFLOW

Um olhar sobre o esmagamento de buffer

>>> Um codigo malicioso pode ser executado somente se fornecido pelo atacante!(?)

>>> Para executar código malicioso, o atacante precisa sobrescrever o endereço de retorno das funçoes!(?)

>>> Somente buffer em pilha é vulnerável a falhas de segurança que levam a execução de código malicioso!(?)

BUFFER OVERFLOW

Técnicas avançadas

>>> arc injection

>>> pointer subterfuge

>>> heap smashing

BUFFER OVERFLOW

Arc injection

>>> Consiste em injetar uma "linha de comando" maliciosa no lugar de usar injeção de código em buffers.

>>> Utilizado para burlar programas que implementam proteção de memória (separação da memória em área de dados e área de código.)

BUFFER OVERFLOW

Pointer subterfuge

>>> Consiste em modificar o valor de algum ponteiro.

>>> Pode ser usado para:

>>> Modificar o ponteiro de alguma função dinâmica, apontando-o para código fornecido pelo atacante

>>> Modificar o ponteiro de dados de algum objeto dinâmico, utilizando-o para inserir dados arbitrários no programa

>>> Burlar sistemas de proteção contra *buffer overflow*

>>> Modificar o ponteiro de alguma entrada *VPTR (virtual function table)*, também apontando-o para alguma função arbitrária que o atacante deve ter inserido.

BUFFER OVERFLOW

Heap smashing

>>> Consiste em, conjuntamente com um ataque de *VPTR smashing*, introduzir ponteiros para código malicioso no programa.

BUFFER OVERFLOW

Considerações finais!

>>> Muitas técnicas surgiram afim de dificultar a exploração de falhas na verificação de limites de buffer!

>>> A MAIORIA FALHOU!(senão todas!)

>>> Ninguém explora uma falha se ela **não** existe!

Portanto:

>>> A melhor forma de prevenir o comprometimento de seu sistema é atacando **a raíz** do problema: FALHAS NA VERIFICAÇÃO DE LIMITES DO BUFFER

ONDE ESTAMOS

Roteiro do Tutorial

Introdução

Segurança e projeto de software

Introdução sobre ataques

Princípios da programação segura

Buffer Overflow

Condições de corrida

Criptografia

Auditoria de software

Conclusões

CONDIÇÕES DE CORRIDA

Considerações iniciais

>>> Provocada por falhas de algorítmos que utilizam threads

>>> *Threads?*

PARALELO - *THREADS*

Conceito

>>> Threads são processos, concorrentes ou paralelos, usados em diversos tipos de algorítmos.

Exemplo:

```
conecta
enquanto conectado
    em paralelo
        recebe mensagem
        envia mensagem
```

PARALELO - *THREADS*

Característica Atômica

>>> Uma parte de um processo pode ser considerada atômica quando ocorre inteiramente sem interrupções.

>>> Para garantir a característica atômica da parte desejada do processo, se faz necessário o uso de semáforos, ou, caso o sistema operacional permita, uso de interrupções.

CONDIÇÕES DE CORRIDA

Como ocorre

>>> Existe no software uma situação inesperada e indesejada, que dificilmente ocorre, apenas em uma janela de tempo muito pequena, mas que pode ser perfeitamente explorável pelos atacantes

>>> Pode ocorrer tanto entre processos concorrentes no software quanto na interação entre software e sistema operacional.

CONDIÇÕES DE CORRIDA

Porque ocorre

>>> Erros no desenvolvimento de agorítmos concorrentes, que geram situações inesperadas e indesejadas.

>> Geralmente, ocasionados pela suposição que determinada parte do processo é atômica, quando na verdade não é.

CONDIÇÕES DE CORRIDA

Principais Consequencias

>>> Pode permitir indevidamente a alteração de arquivos.

>>> Pode alterar indevidamente as permissões de determinado arquivo.

>>> Pode gerar uma situação onde a autenticação em um sistema possa ser transposta.

CONDIÇÕES DE CORRIDA

Considerações finais

>>> Programação concorrente é um tipo de programação extremamente dificil de ser debugada! Erros podem existir, e estes erros podem ser de difícil correção.

Portanto

>>> Sempre prefira o uso de um único processo quando a operação a ser realizada seja crítica!

ONDE ESTAMOS

Roteiro do Tutorial

Introdução

Segurança e projeto de software

Introdução sobre ataques

Princípios da programação segura

Buffer Overflow

Condições de corrida

Criptografia

Auditoria de software

Conclusões

CRIPTOGRAFIA

Introdução - recordando:

O que é criptografia?

>>> Ato de **embaralhar** as informações com determinada **chave,** de forma que somente o detentor da chave consegue reorganizar a informação.

>>> Existem 2 tipos principais de criptografia: as que utilizam chave simétrica e as que utilizam chave assimétrica

CRIPTOGRAFIA

Objetivos

>>> Deixar uma mensagem ilegível, a não ser para aqueles que possuam a chave de decriptografia.

>>> Confere CONFIDENCIALIDADE ao software

>>> Garantir que uma determinada mensagem não foi modificada, utilizando algorítmos de hashing criptográfico

>>> Confere INTEGRIDADE ao software

>>> Determinar a identidade de determinado usuário, utilizando o recurso de chaves assimétricas assinadas.

>>> Confere AUTENTICIDADE ao software

CRIPTOGRAFIA

Pilares da segurança

>>> CONFIDENCIALIDADE

>>> INTEGRIDADE

>>> AUTENTICIDADE

A criptografia pode ser considerada a panacéia da segurança?

"Não é suficiente nos protegermos com leis,
temos que nos proteger com matemática."

Bruce Schneier, em *Applied Cryptography*.

CRIPTOGRAFIA

Análise dos fatos

"Há sete anos, escrevi outro livro: *Applied Cryptography.* Nele, descrevi uma utopia matemática: algoritmos que manteriam os segredos mais profundos protegidos por milênios, protocolos que poderiam realizar as interações mais fantásticas – jogos e apostas sem regulamentação, autenticação indetectável, dinheiro anônimo – de forma segura e protegida.

(...)

Mas isso não é verdade. A criptografia não pode fazer nada disso."

***Bruce Schneier**, em Secrets & lies. (Segurança.com)*

CRIPTOGRAFIA

Análise dos fatos - II

"Desde que escreví o livro, tenho me mantido como consultor de criptografia.(...)

Para minha surpresa inicial, descobrí que os pontos fracos não tinham nada a ver com a matemática. Eles estavam no hardware, **no software**, nas redes e nas pessoas."

***Bruce Schneier**, em Secrets & lies. (Segurança.com)*

"Segurança é uma corrente: tão forte quanto seu elo mais fraco"

CRIPTOGRAFIA

Tópicos

>>> Determinismo X aleatoriedade

>>> Aplicação de algorítmos criptográficos

>>> Entropia

CRIPTOGRAFIA

Determinismo X aleatoriedade

>>> Determinismo: De condição determinada.

>>> "Previsível"

>>> Aleatóriedade: De Condição aleatória.

>>> "Imprevisível"

Computadores são máquinas determinísticas!

(Não existe aleatorieadade – na teoria!)

CRIPTOGRAFIA

Determinismo X aleatoriedade

Pseudo-aleatoriedade

>>> Utilizado pelos computadores para simular aleatoriedade

>>> Gerado por funções (que são previsíveis)

>>> Que usam **Sementes!**

CRIPTOGRAFIA

Determinismo X aleatoriedade

Dedução na Pseudo-aleatoriedade

>>> Tão fácil quanto deduzir a **Semente!**

>>> Facilitado pelo uso de situações anteriores.

> Vide suposto "Ataque de Kevin Mitnick" a Tsutomu Shimomura

> predileção da sequência de ACK (que deveria ser aleatórea)

CRIPTOGRAFIA

Aplicação de algorítmos criptográficos

>>> CONFIDENCIALIDADE

>>> INTEGRIDADE

>>> AUTENTICIDADE

CRIPTOGRAFIA

Aplicação da criptografia - Confidencialidade

>>> Utilização de ambos os tipos de chaves (PGP)

>>> Chaves assimétricas (preferencialmente assinadas) criptografam uma chave símétrica única

>> chave simétrica precisa ser **aleatória**

>>> o texto a ser protegido é criptografado usando a chave simétrica

>>> A chave simétrica, criptografada pela chave pública, é enviada ao destinatário da mensagem, assim como a mensagem, criptografada pela chave simétrica.

CRIPTOGRAFIA

Aplicação da criptografia - Integridade

>>> Utilização de chave assimétrica (PGP)

>>> O texto passa por um hashing, e é criptográfico utilizando a chave privada do remetente. o resultado disso é chamado *assinatura*

>>> Tanto o texto como a assinatura são enviados ao destinatário

>>> No destinatário, a assinatura é decriptografada, e é aplicado hashing sobre o texto recebido. se o hashing for identico à assinatura, o texto é validado

CRIPTOGRAFIA

Aplicação da criptografia - Autenticação

>>> Utilização de chave assimétrica (PGP)

>>> Um **nounce** é enviada ao usuário pelo servidor.

>>> Este nounce é criptografado pelo usuário, usando a chave privada. O resultado é enviado ao servidor.

>>> No servidor, o resultado é decriptografado usando a chave pública, e caso o texto enviado e o recebido sejam iguais, o usuário é validado

>>> O **nounce** precisa ser aleatório!

CRIPTOGRAFIA

algorítmos criptográficos – Quais usar?

>>> Chaves simétricas: DES, RC4, IDEA, AES, etc...

>>> Algorítmos para hash: SHA-1, MD5, RIPEMD-160, etc...

>>>Chaves assimétricas: RSA, El Gamal, DSA, etc...

>>> NUNCA crie seu próprio algoritmo criptográfico! Programadores não são criptoanalistas, portanto dificilmente terão conhecimento técnico suficiente para poder criar seu próprio algorítmo.

CRIPTOGRAFIA

algorítmos criptográficos – Como usar?

>>> Bibliotecas prontas são a melhor alternativa

>>> Algorítmos criptográficos são difíceis de implementar

>>> O que a comunidade já usou e analizou provavelmente terá menos falhas de segurança do que algo que apenas uma pessoa (ou um pequeno grupo de pessoas) usou e analizou.

CRIPTOGRAFIA

algorítmos criptográficos – Bibliotecas.

>>> **Cryptlib** – Para linguagem C

>>>www.cs.auckland.ac.nz/~pgut001/cryptlib/

>>> **OpenSSL** – Para linguagem C – pode ser chamada por *shell-scripts* através de um programa

>>>www.openssl.org

>>> **Crypto++** - Para C++

*>>> www.eskimo.com/~weidai/cryptlib.html

>>> **BSAFE** – C e JAVA

>>>www.rsasecurity.com

CRIPTOGRAFIA

mineração de entropia

>>> O que é mineiração de entropia?

>>> É o ato de coletar informações aleatórias de forma a gerar um **número (ou sequência de números) aleatório**.

>>> Extremamente importante na geração do NOUNCE, para autenticação, e na geração da CHAVE SIMÉTRICA, para confidencialidade. Além de ser necessário para a criação do PAR DE CHAVES ASSIMÉTRICAS.

CRIPTOGRAFIA

mineração de entropia

>>> Como é feita?

>>> Captando informações de hardware específico.

>>> Analizando eventos aleatórios (digitação e chegada de pacotes, por exemplo)

CRIPTOGRAFIA

mineração de entropia

>>> Como fazer?

>>> Alguns sistemas operacionais fazem por você!

>>> No caso do linux, use /dev/random

>> NUNCA use /dev/urandom

CRIPTOGRAFIA

mineração de entropia

>>> O Windows não minera entropia! O processo tem que ser manual.

>>> No caso, seja **pessimista** em relação aos calculos para a mineração de entropia. Antes pecar pelo excesso (e falta de performance) do que pecar pela falta (e falta de segurança.)

ONDE ESTAMOS

Roteiro do Tutorial

Introdução

Segurança e projeto de software

Introdução sobre ataques

Princípios da programação segura

Buffer Overflow

Condições de corrida

Criptografia

Auditoria de software

Conclusões

AUDITORIA DE SOFTWARE

Para que serve?

>>> Descobrir falhas de segurança.

>>> Descobrir *Backdoors*.

>>> Certificar um software.

AUDITORIA DE SOFTWARE

Quem deve auditorar?

>>> Deve ser escalado uma pessoa (ou um grupo de pessoas) específicamente para auditorar o código.

>>> Preferêncialmente, os auditores não devem ter contato com a fase de desenvolvimento.

>> Cabeça livre de vícios.

>> Impossibilidade de ocultamento de falhas propositais.

AUDITORIA DE SOFTWARE

Como deve ser efetuada a auditoria?

>>> Dividido em fases:

>> Análise do algorítmo

>> Análise do código

AUDITORIA DE SOFTWARE

Análise do algorítmo

>>> Dividido em fases:

>> Mineração de informações

>> Análise

>> Relatórios

AUDITORIA DE SOFTWARE

Análise do algorítmo

>>> Mineração de informações

> Analizar os requerimentos do sistema(Software e Segurança).

> Entender o algorítmos utilizados.

> Sanar as dúvidas em relação ao projeto.

AUDITORIA DE SOFTWARE

Análise do algorítmo

>>> Análise

>> Procurar por formas de quebrar o sistema, usando “Árvores de Ataque”

>> Analizar a factibilidade dos ataques.

AUDITORIA DE SOFTWARE

Exemplo de "Arvores de Ataque"

Interceptar uma conexão SSH

1. Quebrar a criptografia
    - 1.1 Quebrar a criptografia da chave pública
        - 1.1.1 Usando RSA?
            - 1.1.1.1 Fatorar os módulos
            - 1.1.1.2 Achar vulnerabilidade na implementação
            - 1.1.1.3 Achar um novo ataque ao sistema criptográfico
        - 1.1.2 Usando El Gamal?
            - 1.1.2.1 Achar uma falha na implementação
    - 1.2 Quebrar a criptografia da chave simétrica
    - 1.3 Quebrar o uso de criptografia no protocolo
2. Obter a chave

4. Executar ataque homem-do-meio

AUDITORIA DE SOFTWARE

Análise de “Arvores de Ataque”

>>>Dificil de ser construida(é mais uma arte do que uma ciência)

> É necessário ter vasto conhecimento de ataques possíveis

> É necessário ter vasto conhecimendo do software a ser analizado

> É nessessário saber como, onde e quando aprofundar em detalhes na árvore.

AUDITORIA DE SOFTWARE

Relatório dos Fatos

>>> Traduzir a análise da árvore em relatórios

>> Detalhar as falhas de segurança encontradas

>> Se possível, sugerir correções

AUDITORIA DE SOFTWARE

Análise de código

>>>Procurar por erros da fase de codificação

>> estouro de buffer, tratamento de expressões, etc...

>>>Procurar por implementações errôneas de algorítmos

>> Condições de corrida e deadlocks por uso incorreto (ou não uso) de semáforos para acesso de regiões críticas; implementações não fiéis ao algorítmos, etc...

>>>Procurar por *backdoors*

AUDITORIA DE SOFTWARE

Análise de código

>>> Foco na comunicação entre programa e fontes não confiáveis

>> Entrada padrão

>> Arquivos

>> Bibliotecas alheias

>> Hardware

>> Etc...

AUDITORIA DE SOFTWARE

Ferramentas de Auxílio

>>>RATS

>> Utilizado para análise de código em Perl, Pyton, C e Php

>> www.securesw.com/rats/

>>>Flawfinder

>> Utilizado para análise de código em C/C++

>> www.dwheeler.com/flawfinder/

AUDITORIA DE SOFTWARE

Ferramentas de Auxílio

>>> Nunca devem ser a única forma de análise em código

> Auxiliam na detecção de falhas, mas não provam que um sistema é seguro.

> Grande quantidade de Falsos Positivos.

ONDE ESTAMOS

Roteiro do Tutorial

Introdução

Segurança e projeto de software

Introdução sobre ataques

Princípios da programação segura

Buffer Overflow

Condições de corrida

Criptografia

Auditoria de software

Conclusões

CONSIDERAÇÕES FINAIS

Programação Segura

>>> Falhas em software são as raízes da insegurança

>>> Programar não é apenas codificar

>>> Não é dada devida enfase em segurança nos cursos e livros de programação.

CONSIDERAÇÕES FINAIS

Desenvolvimento de softwares

>>> Requisitos de software vão de encontro com os requisitos de segurança.

>>> Segurança em software é análize de risco

CONSIDERAÇÕES FINAIS

Falhas em software

>>> Buffer overflow não ocorre se existir a correta checagem de limites em buffers.

>>> Condições de corrida não ocorrem se o trabalho com threads for efetuado de maneira cautelosa e consiente.

CONSIDERAÇÕES FINAIS

Criptografia

>>> Criptografia não é a panacéia da segurança. Não podemos confiar cegamente nela.

>>> NUNCA devemos inventar nossos algorítmos criptográficos

>>> Se possível, é preferivel utilizar de bibliotecas conhecidas a implementar a sua versão do algorítmo criptográfico

CONSIDERAÇÕES FINAIS

Auditoria de código

>>> Importante para, além de checar a existência ou não de falhas de segurança no software, garantir a não inclusão de BackDoors no código e conferir credibilidade em relação a segurança de software

>>> O auditor de software, preferencialmente, não deve ser o responsavel pela implementação do software.

CONSIDERAÇÕES FINAIS

Por Fim:

K.I.S.S.

Keep It Simple, Stupid

Mantenha as coisas de forma simples

(onde simples != simplório)

CONSIDERAÇÕES FINAIS

Relembrando:

Nós não precisaríamos gastar tanto tempo, dinheiro e esforço em segurança de redes se não tivéssemos uma segurança em software tão ruim

LEITURAS RECOMENDADAS

Building Secure Software – How to avoid Security Problens the Rigth Way

>> **Autores:** John Viega & Gary McGraw

>> **Editora:** Addison-Wesley **ISBN:** 0-201-72152-X

Secure Coding – Principles & Practices

>> **Autores:** Mark G. Graff & Kenneth R. van Wyk

>> **Editora:** O'REILLY **ISBN:** 0-596-00242-4

Secure Programming Cookbook – for C and C++

>> **Autores:** John Viega & Matt Messier

>> **Editora:** O'REILLY **ISBN:** 0-596-00394-3