Curso de Extensão

Tecnologias Microsoft

UNICAMP

INF-0998

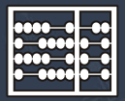
Programação segura (Segurança de software) Aula 3

Prof. Dr. Alexandre Braga, CISSP, CSSLP, PMP

alexbraga@ic.unicamp.br / ambraga@cpqd.com.br / braga.alexandre.m@gmail.com

br.linkedin.com/in/alexmbraga

26 de Novembro de 2022



INSTITUTO DE COMPUTAÇÃO

Regras de ouro do software seguro

O que são as regras de ouro?

Boas práticas genéricas de segurança de sistemas

- Boas práticas conhecidas por vários nomes
- Princípios de projeto seguro
- Boas práticas de segurança de sistemas
- Regras de ouro do software seguro

- Vistas primeiro em SALTZER & SCHROEDER, 1975
- Saltzer, J.H. and Schroeder, M.D., 1975. *The protection of information in computer systems*. Proceedings of the IEEE, 63(9), pp.1278-1308.
- http://web.cs.wpi.edu/~guttman/cs557_website/papers/saltzer1975.pdf

Economia de mecanismos

Simplicidade de implementação

O projeto de segurança deve ser simples e pequeno

- Sendo simples, contém menos vulnerabilidades
- Sendo pequeno, pode ser verificado manualmente

Siga o caminho mais simples!

- *Keep It Simple, Stupid* (KISS)!
- Keep It Simple and Short
- Ex.: arquiteturas de microsserviços

Projeto simples é mais fácil de entender e corrigir

- Vulnerabilidades são comuns em software complexo
- Verificações localizadas, mais fácil de testar
- Menos funcionalidade = menos exposição

Cuidado com os pontos de estrangulamento (gargalos)

- Software centralizado pelo qual passa todo acesso

Falha segura

Prever e planejar salvaguardas para falhas do sistema

Quando em falha, o sistema sempre adota o estado mais seguro

Exemplo de falha segura: autorização e controle de acesso

- Acesso baseado em permissões e não em proibições
- Acesso negado quando não é possível verificar permissão
- Negue por padrão!

Exemplo de falha segura: Elevadores

- Projetados com a expectativa da falha de energia
- Em um apagão, fica preso aos cabos ou aos *guide rails*

Exemplo de falha segura: Sistema de firewall

- Ex.: Se o firewall falha, nada entra (princípio *deny all*)
- Atacante não é incentivado a causar a falha

Mediação completa

Componente “monitor de referência” vigia todos os acessos

Todo acesso deve ser verificado

- a todo recurso computacional
- a todo dado sensível
- a todo ativo de valor

Caminho confiável entre o usuário e o ativo de valor

- Sempre possível verificar as permissões de acesso
- Atalhos para admins (backdoors) quebram este princípio

Exemplos ruins:

- Porta dos fundos sem vigilância predial
- Conta administrativa com senha embutida no código
- Aplicação com modo “desenvolvedor” ou “depuração”

Projeto aberto

Tecnologias padronizadas, avaliadas publicamente

A segurança não deve ser baseada em segredos de construção e nem em tecnologias proprietárias

Evitar a segurança por obscuridade

- Fui eu que fiz, então está seguro? ***NÃO!***
- Ninguém viu e nem conhece, então está seguro? ***NÃO!***
- A obscuridade tem eficácia questionável e utilidade duvidosa

Exemplos de segurança por obscuridade

- Programas Java, dotNET, JS são descompilados
- Segredos hardcoded no fonte ou binário são revelados

- Uso de criptografia proprietária e implementações domésticas
- Segurança é comprometida pela engenharia reversa de código

Responsabilidades separadas

Separação de responsabilidades e de privilégios

A criação de usuários todo-poderosos deve ser evitada

- Um único usuário nunca dever ser capaz de realizar sozinho todos os passos de uma tarefa importante
- Ele não age em benefício próprio às escondidas

Para inibir a fraude, a responsabilidade sobre uma tarefa complexa deve ser dividida entre dois ou mais usuários

- A fraude só é possível pela conspiração ou conluio entre as partes envolvidas (quadrilhas)

Exemplos de responsabilidades divididas:

- Dupla custódia de senhas administrativas
- Etapas de aprovação em um processo ou workflow
- Revisão de código obrigatória antes do *commit*

Privilégios mínimos

Este princípio complementa o anterior

- O usuário deve possuir apenas as permissões suficientes para a realização das tarefas necessárias a sua função na organização
- O usuário com permissões insuficientes é ineficiente, pois não consegue realizar suas funções
- O usuário com permissões em excesso expõe os recursos da organização a riscos desnecessários
- O princípio se aplica também aos usuários (contas) de serviço
- Adote o princípio de privilégios mínimos no projeto das estórias de usuários e dos casos de uso!
- Exemplos de privilégios excessivos desnecessários
- Manobrista do estacionamento não pode abrir o porta-luvas
- Admin do servidor web não pode modificar os HTMLs da aplicação

Interseção reduzida

Não compartilhar informações de segurança entre usuários

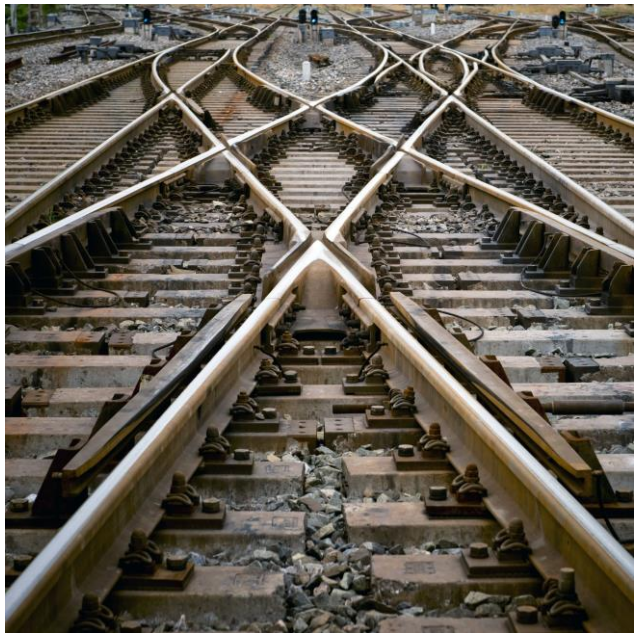
Mínimo denominador comum

- O compartilhamento de credenciais e informações de segurança entre usuários deve ser reduzido evitado
- Evitar compartilhamento de senhas, chaves criptográficas, segredos e outros parâmetros de segurança

Este compartilhamento dificulta a autenticação inequívoca e a responsabilização do usuário

Exemplos:

- Reuso de tokens repetidos por sessões/usuários diferentes
- Conta administrativa com a senha compartilhada



Defesa em profundidade

Segurança em camadas

Redundância e diversidade na proteção

- Camadas diferentes de defesa que se complementam
- Não confiar em apenas um único mecanismo de proteção
- Sistemas diferentes mas com a mesma função

Exemplos:

- Dinheiro no banco: guardas, cofres, alarmes, tinta, etc.
- Armadilhas e desafios em Indiana Jones e o Cálice Sagrado

- Diferentes SOs evitam hacking de todos os endpoints
- Outro firewall de outro fabricante entre BD e servidor web

Desvantagens:

- Necessita de especialistas em todas as tecnologias
- Custo operacional X segurança extra

Proteção do elo mais fraco

Mais cuidado com o componente de maior risco

"Uma corrente é tão forte quanto seu elo mais fraco"

"Um software é tão seguro quanto seu componente mais vulnerável."

A segurança é desequilibrada (desbalanceada) em construções que usam muitos componentes diferentes

Exemplos de elos/componentes fracos:

- Portas de acesso indevido (backdoor)
- Senhas fracas (fáceis de adivinhar)
- Pessoas suscetíveis à engenharia social
- Vulnerabilidades de software expostas (ex.: buffer overflow, XSS, SQLi)

Default (padrão) seguro

Segurança “Plug&Play”

Default seguro facilita a implantação e operação seguras

- Quanto mais funções, mais chance de exploração
- Quanto mais funções, maior a superfície de ataque

Robustecimento (“Hardening”) de sistemas

- Todos os serviços desnecessários são desligados
- Habilitar somente as funções simples mais usadas
- Funções avançadas e pouco usadas ficam desligadas

Exemplos ruins:

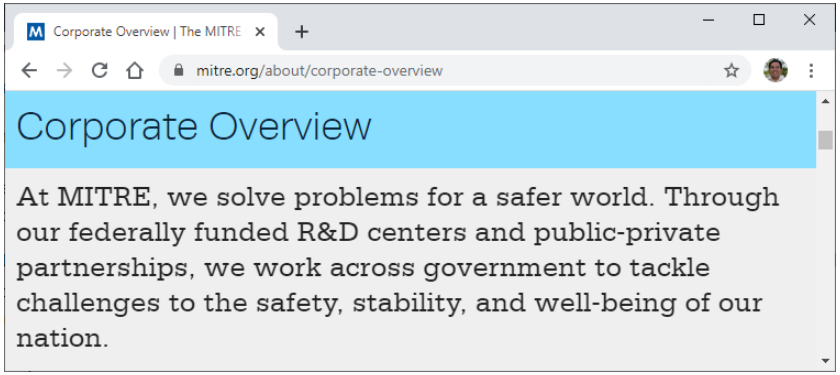
- Equipamentos domésticos “Plug&Play” com segurança desligada
- Como esses: roteadores WiFi, Impressoras wireless, SmartTVs, etc.

Vulnerabilidades de software

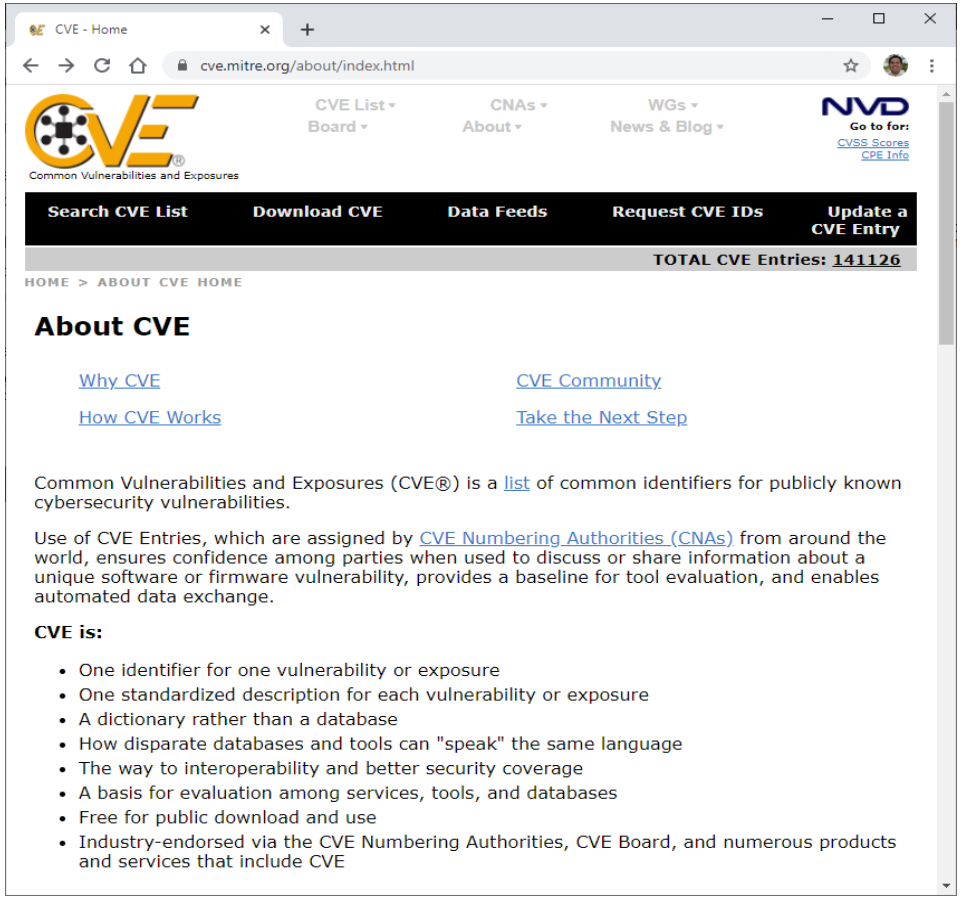
Mitre e CVE

Common Vulnerabilities and Exposures (CVE)

- Mitre é a organização que mantém as listas. Já o CVE é a lista de vulnerabilidades exploradas.



https://www.mitre.org/about/corporate-overview

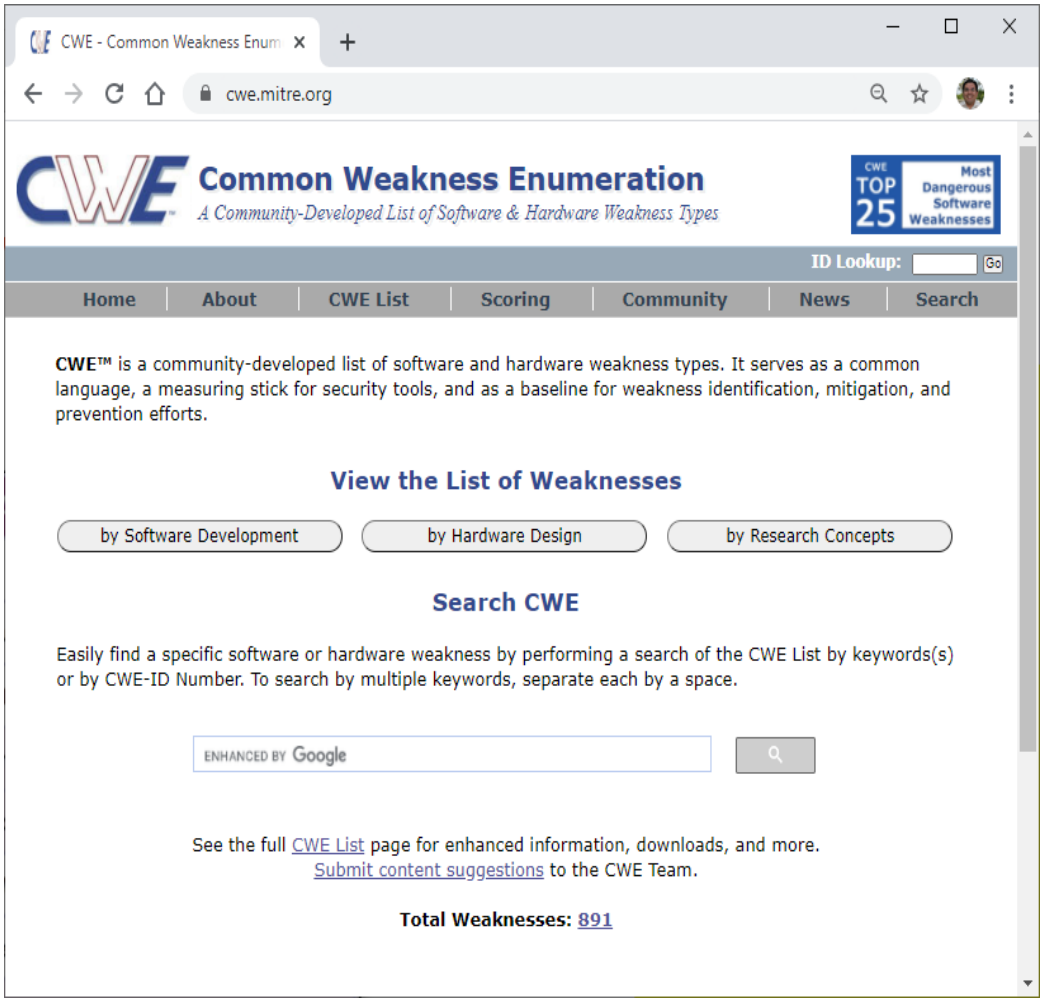


https://cve.mitre.org/about/index.html

Mitre e CWE

Common Weakness Enumeration (CWE)

- https://cwe.mitre.org/
- CWE é a lista de fraquezas que podem (ou não) ter sido exploradas
- Ao lado, as visões com pontos de vista diferentes do CWE



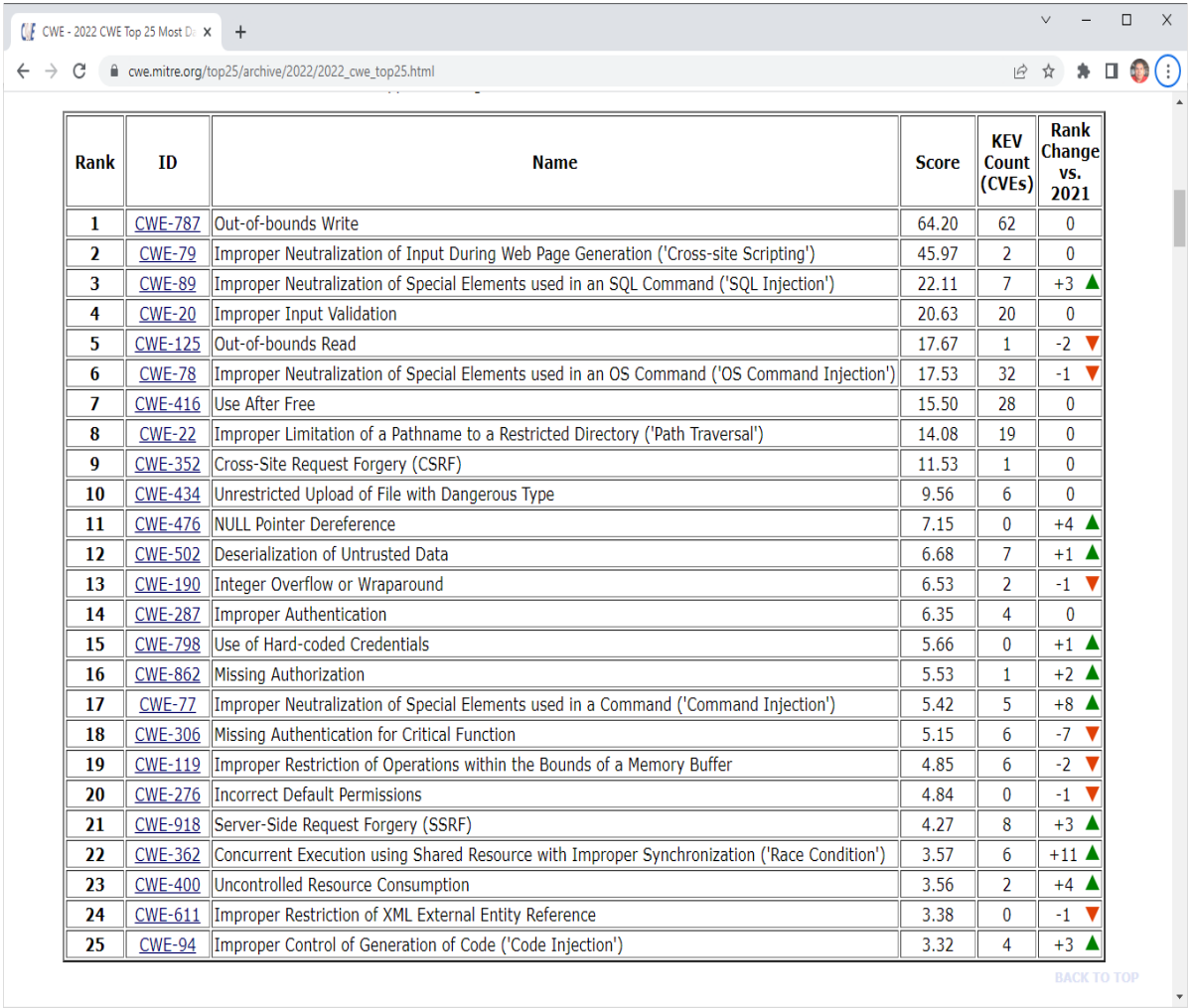
Visão CWE Top 25 (2022)

2022 CWE Top 25 Most Dangerous Software Weaknesses

- https://cwe.mitre.org/top25
- CWE é a lista de fraquezas que podem ou não ter sido exploradas
- **Visões com pontos de vista do CWE**

Instruções para os grupos:

- Escolher duas fraquezas
- Estudar as fraquezas
- Explicar como a fraqueza aparece



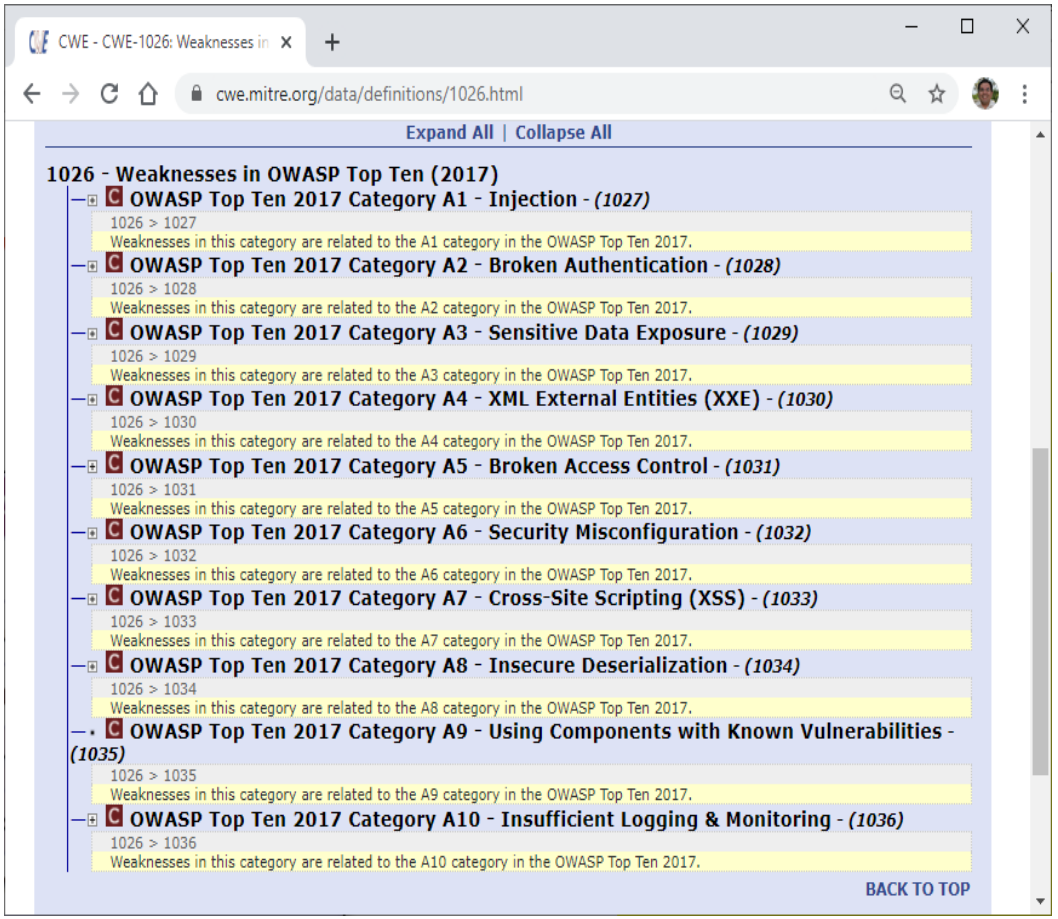
Rank	ID	Name	Score	KEV Count (CVEs)	Rank Change vs. 2021
1	CWE-787	Out-of-bounds Write	64.20	62	0
2	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	45.97	2	0
3	CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')	22.11	7	+3 ▲
4	CWE-20	Improper Input Validation	20.63	20	0
5	CWE-125	Out-of-bounds Read	17.67	1	-2 ▼
6	CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')	17.53	32	-1 ▼
7	CWE-416	Use After Free	15.50	28	0
8	CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')	14.08	19	0
9	CWE-352	Cross-Site Request Forgery (CSRF)	11.53	1	0
10	CWE-434	Unrestricted Upload of File with Dangerous Type	9.56	6	0
11	CWE-476	NULL Pointer Dereference	7.15	0	+4 ▲
12	CWE-502	Deserialization of Untrusted Data	6.68	7	+1 ▲
13	CWE-190	Integer Overflow or Wraparound	6.53	2	-1 ▼
14	CWE-287	Improper Authentication	6.35	4	0
15	CWE-798	Use of Hard-coded Credentials	5.66	0	+1 ▲
16	CWE-862	Missing Authorization	5.53	1	+2 ▲
17	CWE-77	Improper Neutralization of Special Elements used in a Command ('Command Injection')	5.42	5	+8 ▲
18	CWE-306	Missing Authentication for Critical Function	5.15	6	-7 ▼
19	CWE-119	Improper Restriction of Operations within the Bounds of a Memory Buffer	4.85	6	-2 ▼
20	CWE-276	Incorrect Default Permissions	4.84	0	-1 ▼
21	CWE-918	Server-Side Request Forgery (SSRF)	4.27	8	+3 ▲
22	CWE-362	Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition')	3.57	6	+11 ▲
23	CWE-400	Uncontrolled Resource Consumption	3.56	2	+4 ▲
24	CWE-611	Improper Restriction of XML External Entity Reference	3.38	0	-1 ▼
25	CWE-94	Improper Control of Generation of Code ('Code Injection')	3.32	4	+3 ▲

BACK TO TOP

OWASP Top 10 (visão CWE)

OWASP Top 10 - 2017
A1:2017-Injection
A2:2017-Broken Authentication
A3:2017-Sensitive Data Exposure
A4:2017-XML External Entities (XXE) [NEW]
A5:2017-Broken Access Control [Merged]
A6:2017-Security Misconfiguration
A7:2017-Cross-Site Scripting (XSS)
A8:2017-Insecure Deserialization [NEW, Community]
A9:2017-Using Components with Known Vulnerabilities
A10:2017-Insufficient Logging&Monitoring [NEW,Comm.]

https://owasp.org/www-project-top-ten/



https://cwe.mitre.org/data/definitions/1026.html

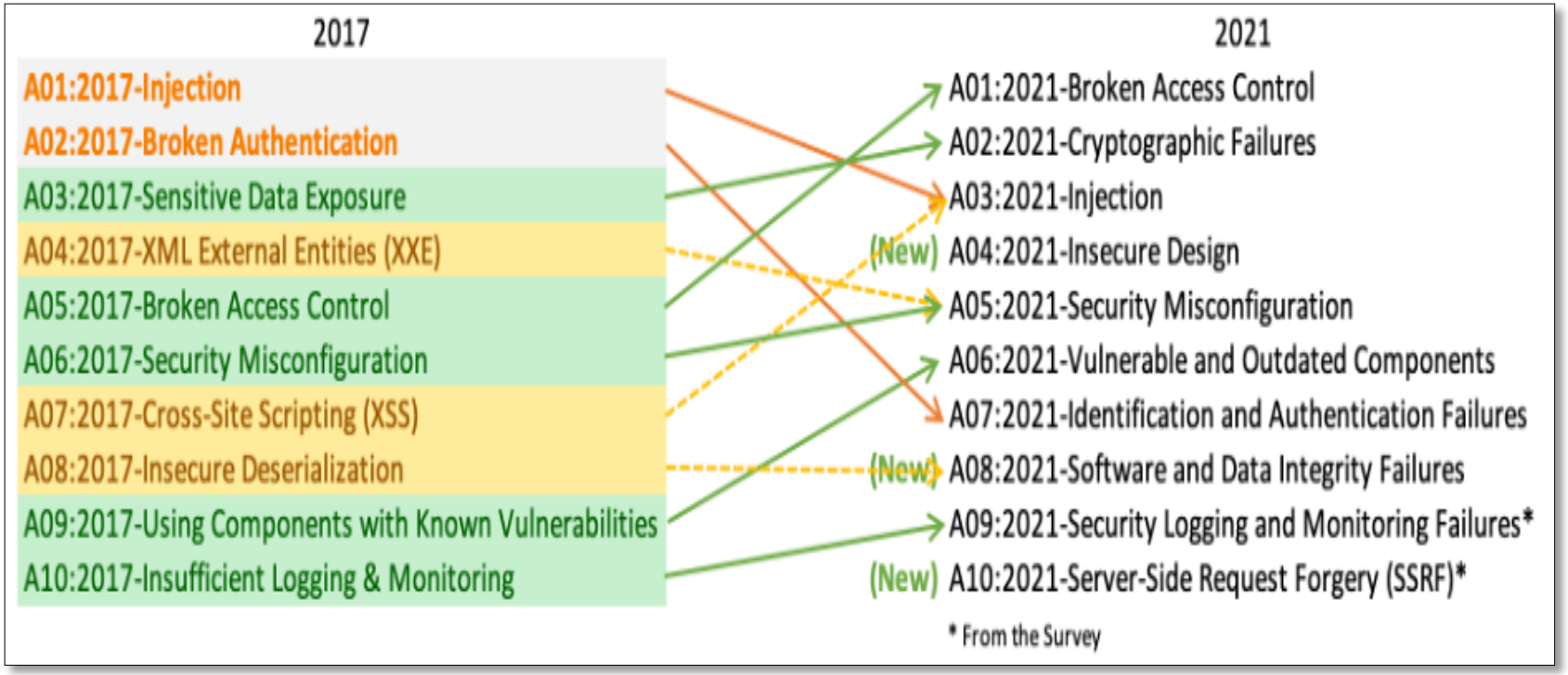
OWASP Top 10 (2017)

https://owasp.org/www-project-top-ten/

A1:2017-Injection	Injection flaws, such as SQL, NoSQL, OS, and LDAP injection, occur when untrusted data is sent to an interpreter as part of a command or query. The attacker's hostile data can trick the interpreter into executing unintended commands or accessing data without proper authorization.
A2:2017-Broken Authentication	Application functions related to authentication and session management are often implemented incorrectly, allowing attackers to compromise passwords, keys, or session tokens, or to exploit other implementation flaws to assume other users' identities temporarily or permanently.
A3:2017-Sensitive Data Exposure	Many web applications and APIs do not properly protect sensitive data, such as financial, healthcare, and PII. Attackers may steal or modify such weakly protected data to conduct credit card fraud, identity theft, or other crimes. Sensitive data may be compromised without extra protection, such as encryption at rest or in transit, and requires special precautions when exchanged with the browser.
A4:2017-XML External Entities (XXE)	Many older or poorly configured XML processors evaluate external entity references within XML documents. External entities can be used to disclose internal files using the file URI handler, internal file shares, internal port scanning, remote code execution, and denial of service attacks.
A5:2017-Broken Access Control	Restrictions on what authenticated users are allowed to do are often not properly enforced. Attackers can exploit these flaws to access unauthorized functionality and/or data, such as access other users' accounts, view sensitive files, modify other users' data, change access rights, etc.
A6:2017-Security Misconfiguration	Security misconfiguration is the most commonly seen issue. This is commonly a result of insecure default configurations, incomplete or ad hoc configurations, open cloud storage, misconfigured HTTP headers, and verbose error messages containing sensitive information. Not only must all operating systems, frameworks, libraries, and applications be securely configured, but they must be patched and upgraded in a timely fashion.
A7:2017-Cross-Site Scripting (XSS)	XSS flaws occur whenever an application includes untrusted data in a new web page without proper validation or escaping, or updates an existing web page with user-supplied data using a browser API that can create HTML or JavaScript. XSS allows attackers to execute scripts in the victim's browser which can hijack user sessions, deface web sites, or redirect the user to malicious sites.
A8:2017-Insecure Deserialization	Insecure deserialization often leads to remote code execution. Even if deserialization flaws do not result in remote code execution, they can be used to perform attacks, including replay attacks, injection attacks, and privilege escalation attacks.
A9:2017-Using Components with Known Vulnerabilities	Components, such as libraries, frameworks, and other software modules, run with the same privileges as the application. If a vulnerable component is exploited, such an attack can facilitate serious data loss or server takeover. Applications and APIs using components with known vulnerabilities may undermine application defenses and enable various attacks and impacts.
A10:2017-Insufficient Logging & Monitoring	Insufficient logging and monitoring, coupled with missing or ineffective integration with incident response, allows attackers to further attack systems, maintain persistence, pivot to more systems, and tamper, extract, or destroy data. Most breach studies show time to detect a breach is over 200 days, typically detected by external parties rather than internal processes or monitoring.

OWASP Top 10 (2021)

Novo link de acesso para a edição de 2021 https://owasp.org/Top10/



Client-side manipulation

- Web application – coleção de programas usadas pelo servidor para responder às requisições do cliente (browser)
  - Geralmente aceite input do usuário: não confie, valide!
- HTTP é stateless, servers não mantém estado
  - Para conduzir transações, web apps tem estado
  - Informação de estado pode ser enviada ao cliente que a ecoa de volta em requisições futuras (p.ex. token de sessão)
- Exemplo de exploração: parâmetros “Hidden” em HTML não são de fato escondidos e podem ser manipulados

Client-side manipulation

```
<form action="/search" method=get>

Search: <input type=text name=search
onChange="javascript:
validateQuery(this.value);">

<input type=submit value=Search>
```

A intenção do programador foi fazer a validação do lado cliente…

Mas faltou a validação do lado servidor...

O atacante acessa a URL diretamente e força dados inválidos para injetar código

```
http://site.com/search?search='something;
select * from users'
```

Nunca confie nos dados de entrada!!!

Variações do mesmo tema!

“Todo input é culpado até provar o contrário”

- Estouros de Buffer (Buffer Overruns)

10101101101101011011	10101101101101011011101000101001010

- SQL Injection

Alexandre	Alexandre’; drop table users --

- Cross-Site Scripting

Alexandre	<script>var i=document.cookie</script>

Falhas de validação de dados

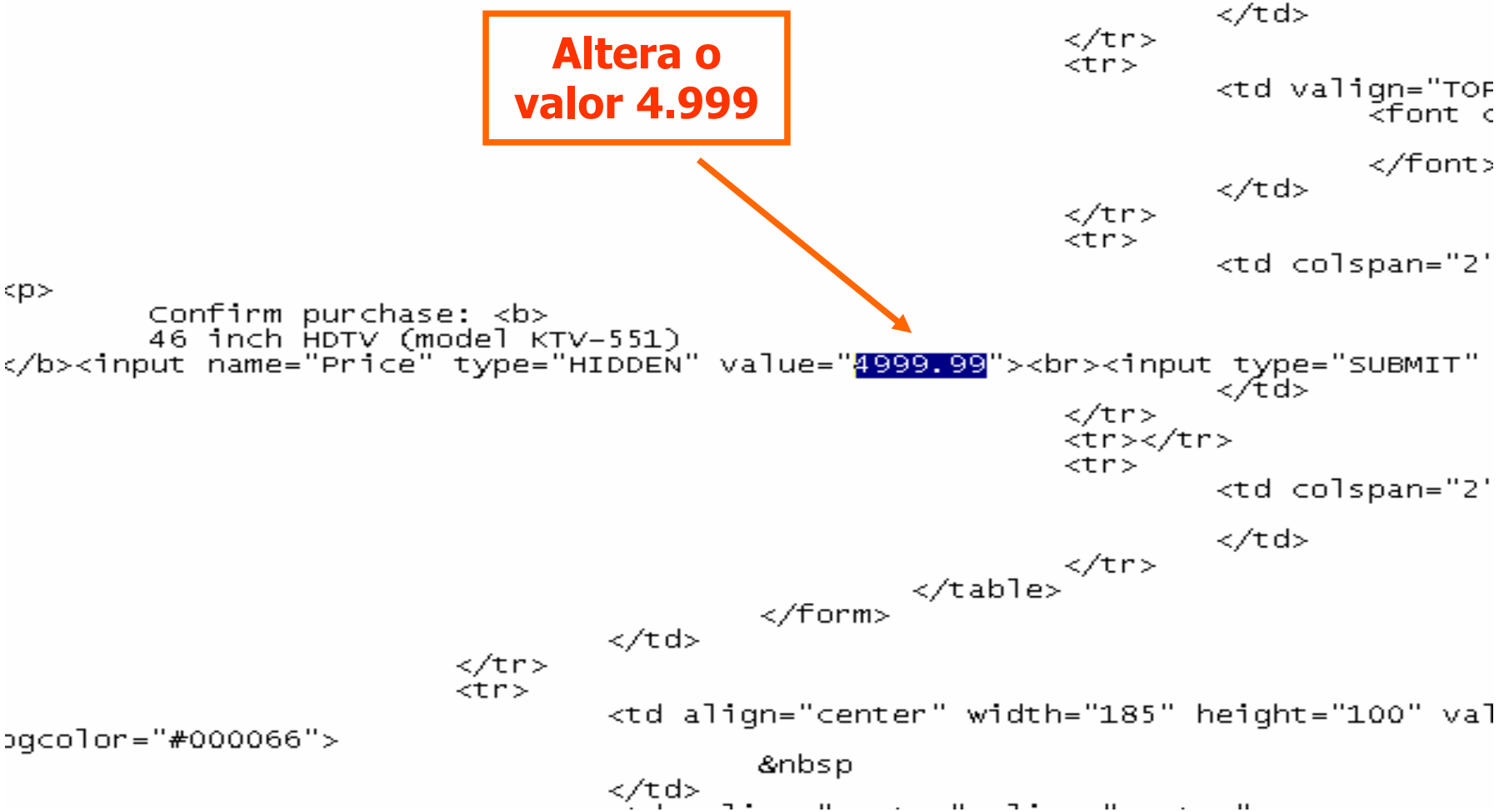
- Manipulação de dados do campo hidden
- (Caso famoso de fraude em e-commerce bastante noticiado)

Click below to confirm your purchase.

Your total price is: **$4999.99**

This amount will be charged to your credit card immediately.

Falhas de validação de dados



Falhas de validação de dados

- Compra uma item muito caro (HDTV) por um preço irrisório

- Conclui o pedido de compra com o valor no browser

Exemplo: Pizza Delivery

Web app para entrega de pizza

- Formulário de pedido online : order.html – um usuário compra uma pizza por $5,50
- Formulário de confirmação: gerado pelo script confirm_order, pede ao usuário para verificar a compra, preço é enviado como hidden form field
- Cumprimento: script submit_order trata o pedido do usuário recebido como requisições GET do formulário de confirmação (variáveis pay & price embutidas como parâmetros na URL)

Pizza Web Site Code

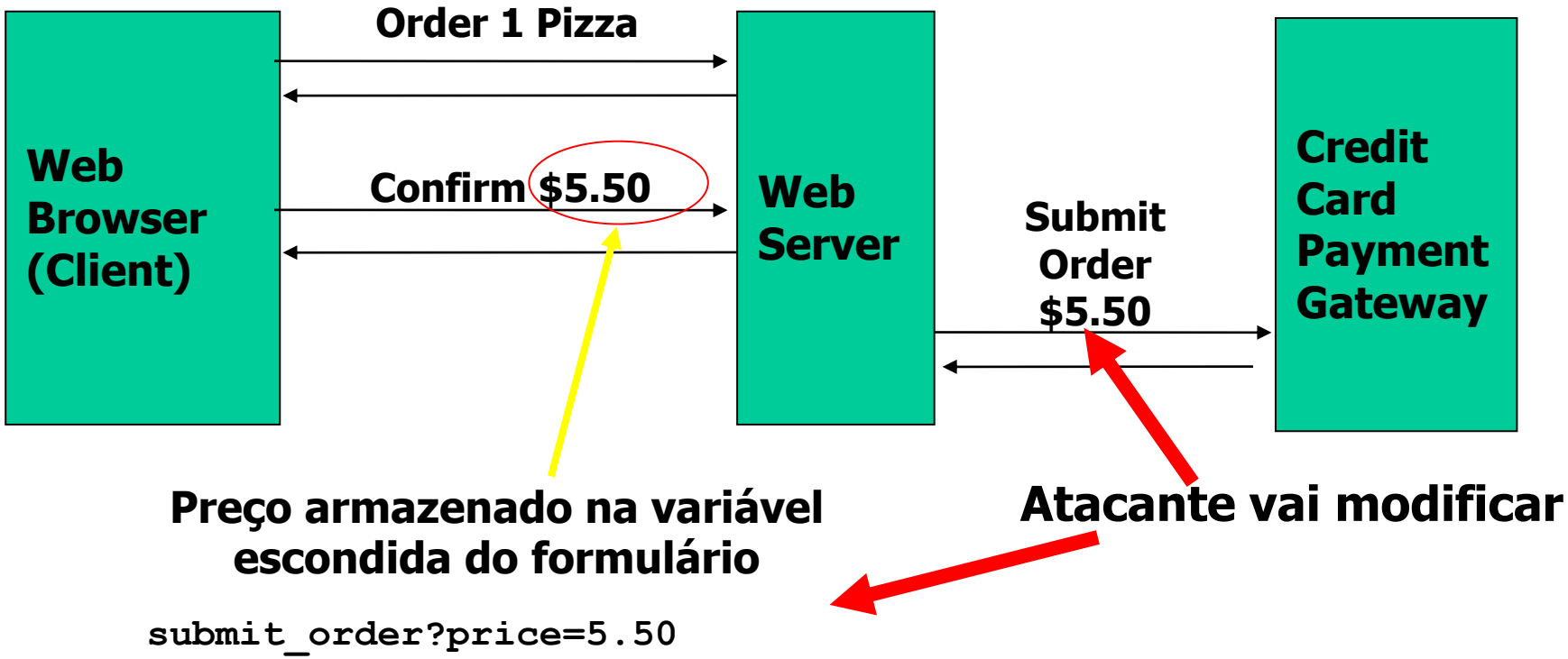
- Formulário de confirmação:

```
<HTML><head><title>Pay for Pizza</title></head>
<body><form action="submit_order" method="GET">
<p> The total cost is 5.50. Are you sure you
would like to order? </p>
<input type="hidden" name="price" value="5.50">
<input type="submit" name="pay" value="yes">
<input type="submit" name="pay" value="no">
</form></body></HTML>
```

- Script de submissão de pedido:

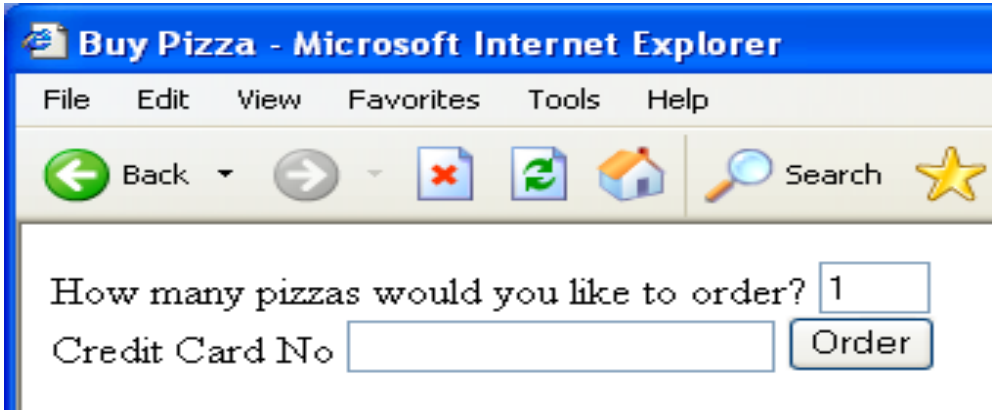
```
if (pay = yes) {
  success = authorize_credit_card_charge(price);
  if (success) {
    settle_transaction(price);
    dispatch_delivery_person();
  } else { // Could not authorize card
    tell_user_card_declined();
  }
} else { display_transaction_cancelled_page(); // no}
```

Exemplo: Comprar Pizza

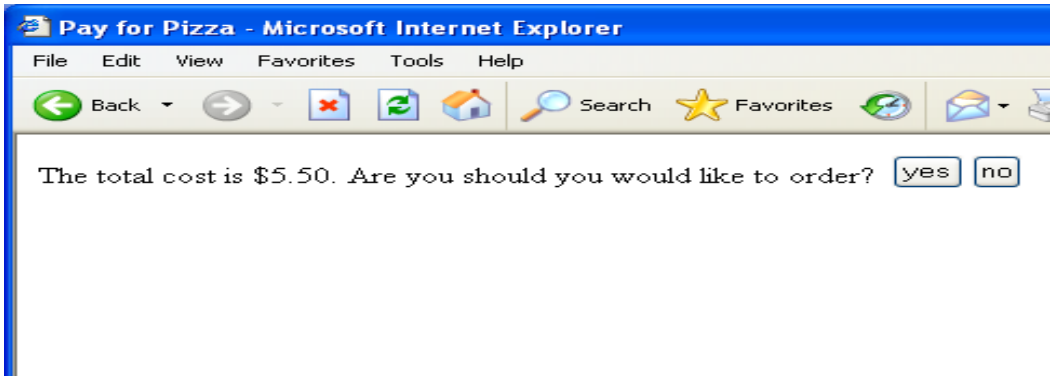


Cenário de ataque

Atacante navega pelo formulário de pedido…



…e submete um formulário de pedido



Cenário de ataque

- E visualiza o código fonte:

```
 total cost is $5.50.
 you should you would like to order?
put type="hidden" name="price" value="5.50">
put type=submit name="pay" value="yes">
put type=submit name="cancel" value="no">
ndv>
```

- Muda o preço no código fonte recarrega a página!

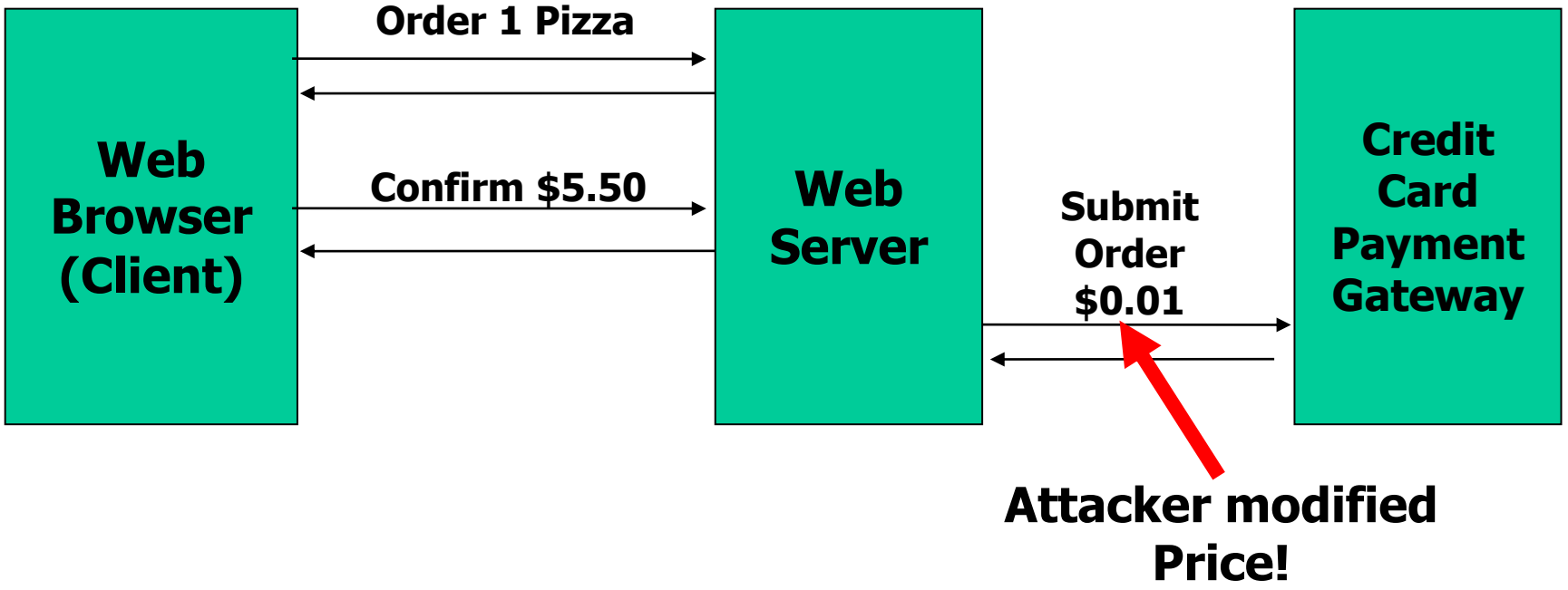
```
Are you should you would like to order?
<input type="hidden" name="price" value="0.01">
<input type=submit name="pay" value="yes">
<input type=submit name="cancel" value="no">
</hndv>
```

- Browser envia a requisição:

```
GET /submit_order?price=0.01&pay=yes HTTP/1.1
```

- Campos Hidden estão em claro

Cenário de ataque



Não é sobre o campo HIDDEN!

É um design defeituoso!

Cenário de ataque

- **Ferramentas de linha de comando para submeter requisições HTTP**
- **curl ou Wget automatizam e aceleram o ataque:**

```
curl https://www.deliver-me-pizza.com/submit_order
?price=0.01&pay=yes
```

- **Mesmo contra POST, podem especificar parâmetros como argumentos para o comando curl ou wget**

```
curl -dprice=0.01 -dpay=yes https://www.deliver-
me-pizza.com/submit_order

wget --post-data 'price=0.01&pay=yes'
https://www.deliver-me-pizza.com/submit_order
```

Resumo Client-side manipulation

- Aplicações web precisam manter estado
  - HTTP é stateless
  - Hidden form fields, cookies, local storage, etc.
  - Gerenciamento de sessão, servidor c/ state…
- Não confie no input do usuário!
  - Manter estado no servidor (space-expensive)
  - Ou assine os parâmetros da transação (bandwidth-expensive)
  - Use cookies, cuidado c/ cross-site attacks
  - Não use JavaScript para computações e validações

Command & SQL Injection

- *Vulnerabilidade de Command injection* – input não confiável inserido na consulta ou no comando
  - String de ataque altera semantica do comando original
  - Ex: *SQL Injection* – dado não saneado usado em consulta ao banco de dados

- SQL Injection Exemplos & Soluções
  - Type 1: Compromete dados de usuário
  - Type 2: Modifica dados críticos
  - Whitelisting & Blacklisting
  - Escaping
  - Prepared Statements and Bind Variables

Como funciona o SQL Injection?

- **Vulnerabilidade comum: login query**
  - SELECT * FROM users
  - WHERE login = **'alex'**
  - AND password = '123'
  - (Faz login se for verdadeiro)

- **Sintaxe de login em um ASP/MS SQL Server**
  - var sql = "SELECT * FROM users
  - WHERE login = '" + *formusr* +
  - "' AND password = '" + *formpwd* + "'";

Injeção de strings

formusr = **' or 1=1 – –**

formpwd = flowers

- **A query final será algo como:**

SELECT * FROM users

WHERE username = **' ' or 1=1**

– – AND password = ‘flowers'

Cenário de ataque (1)

- Ex: Pizza Delivery – Revisão de pedidos
  - Formulário pede número do mês para buscar os pedidos

  - HTTP request:

```
https://www.deliver-me-pizza.com/show_orders?month=10
```

Cenário de ataque (2)

- App constrói consulta SQL a partir de parâmetros:

```
sql_query = "SELECT pizza, toppings, quantity, order_day " +
            "FROM orders " +
            "WHERE userid=" + session.getCurrentUserId() + " " +
            "AND order_month=" + request.getParamenter("month");
```

Normal Query **SQL**

```
SELECT pizza, toppings, quantity, order_day
FROM orders
WHERE userid=4123
AND order_month=10
```

- Type 1 Attack: `month='0 OR 1=1'` !
- encoded URL: (space -> `%20`, = -> `%3D`)

Cenário de ataque (3)

Malicious Query

```
SELECT pizza, toppings, quantity, order_day
FROM orders
WHERE userid=4123
AND order_month=0 OR 1=1
```

Condição WHERE é sempre verdadeira!

OR precede AND

Type 1 Attack: Ganha acesso aos dados privados de outros usuários!

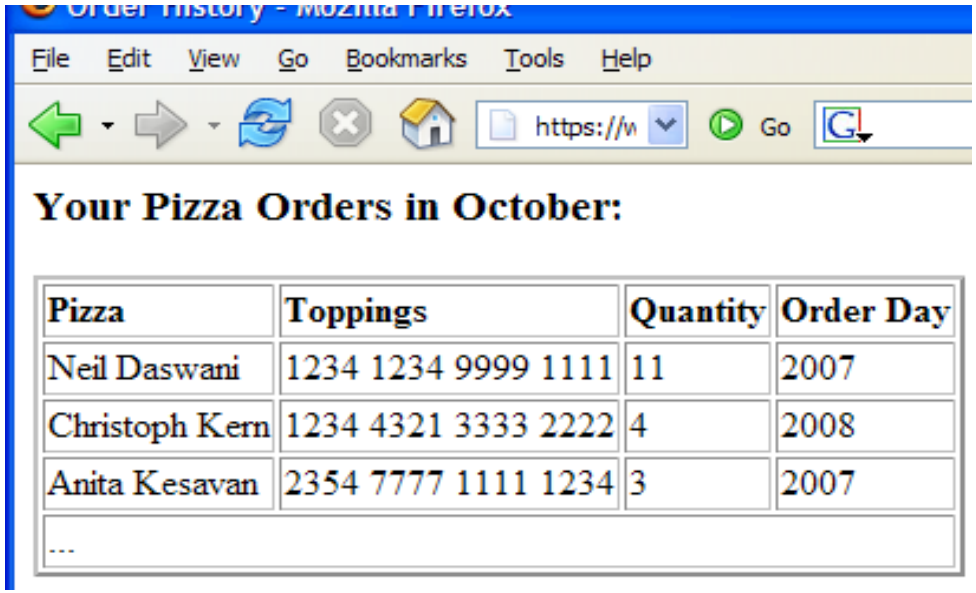
Todos os dados Comprometidos!

Cenário de ataque (4)

- Ataque mais perigoso: atacante faz

```
Month = 0 AND 1=0
UNION SELECT cardholder, number, exp_month, exp_year
FROM creditcards
```

- Atacante é capaz de
  - Combinar 2 consultas
  - 1ª. consulta:
    - Tabela vazia
  - 2ª. consulta:
    - números de cartão



Cenário de ataque (5)

- Pior ainda, atacante faz

```
month=0;
DROP TABLE creditcards;
```

- Então BD executa
  - Type 2 Attack: Remove `creditcards` do schema!
  - Pedidos futuros falham
    - DoS!

```
SELECT pizza, toppings, quantity,
order_day
FROM orders
WHERE userid=4123
AND order_month=0;
DROP TABLE creditcards;
```

- Outras sentenças problemáticas:
  - Modificadoras
    - `INSERT INTO admin_users VALUES ('hacker',...)`
  - Administrativas: shut down DB, OS command…

Cenário de ataque (6)

- Injetando parâmetros: Topping Search

```
sql_query =
  "SELECT pizza, toppings, quantity, order_day " +
  "FROM orders " +
  "WHERE userid=" + session.getCurrentUserId() + " " +
  "AND topping LIKE '%" + request.getParamenter("topping") + "%' ";
```

- Atacante faz:
  - `topping=brzfg%'; DROP table creditcards; --`

- Consulta resultante:
  - SELECT: tabela vazia
  - -- comentario
  - Tabela de cartões é removida

```
SELECT pizza, toppings, quantity,
order_day
FROM orders
WHERE userid=4123
AND topping LIKE '%brzfg%';
DROP table creditcards; --%'
```

Resumo de SQL injection

- SQL injection é um ataque dos mais perigosos
  - Compromete conjuntos completos de dados sensíveis
  - Altera ou danifica dados críticos
  - Dá ao atacante acesso não autorizado ao SGBD

- Usar diversas proteções de modo consistente
  - Whitelisting, input validation & escaping
  - Prepared Statements c/ bind variables

Cross-Site Scripting (XSS)

Descrição

- Atualmente um dos ataques mais populares
- Permite transportar código malicioso até o navegador de outros usuários
- Código é executado no contexto do servidor vulnerável, o que faz com que a política de mesma origem seja respeitada
- Causas:
  - Falta de validação de entrada
  - Falta de tratamento da saída.

- Tipos
- XSS refletido
  - O retorno do ataque acontece imediatamente
- XSS armazenado
  - O retorno do ataque fica latente (armazenado) e acontece em algum momento no futuro
- XSS baseado em DOM
  - Um XML é injetado
- Cross-channel scripting
  - Usa canais diferentes (mobile e web, por exemplo)

Cross-Site Scripting (XSS)

Buffer Overflow

Anatomia de um Buffer Overflow

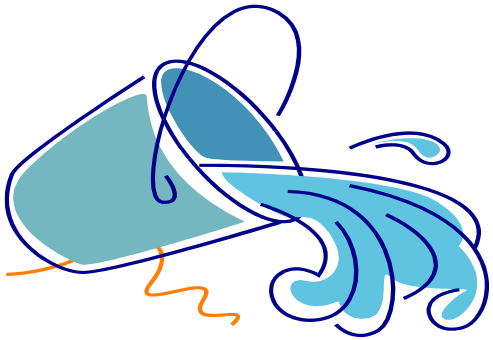
- *Buffer*:
  - Memória usada para armazenar input do usuário
  - Tem tamanho máximo fixo

- *Buffer overflow*:
  - Quanto o input do usuário excede o tamanho máximo do buffer
  - Input extra vai para locais de memória inesperados

Exemplo simples

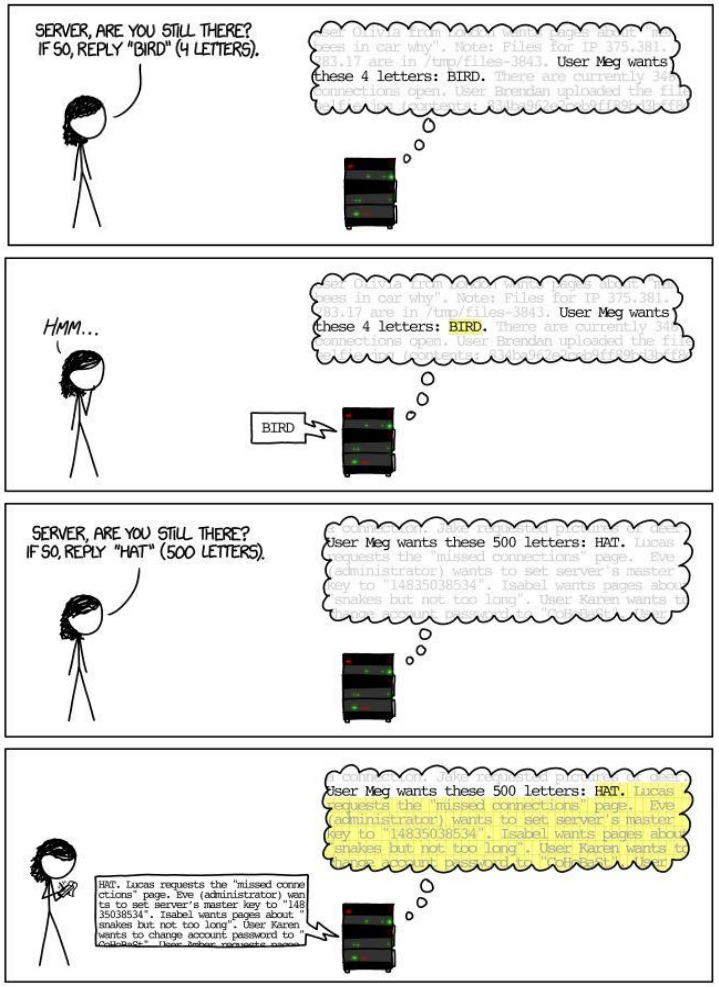
- Usuário malicioso entra > 1024 caracteres, mas variável `buf` só pode armazenar 1024 caracteres
- Caracteres extras transbordam o buffer

```
void get_input() {
    char buf[1024];
    gets(buf);
}
void main(int argc, char*argv[]){
    get_input();
}
```

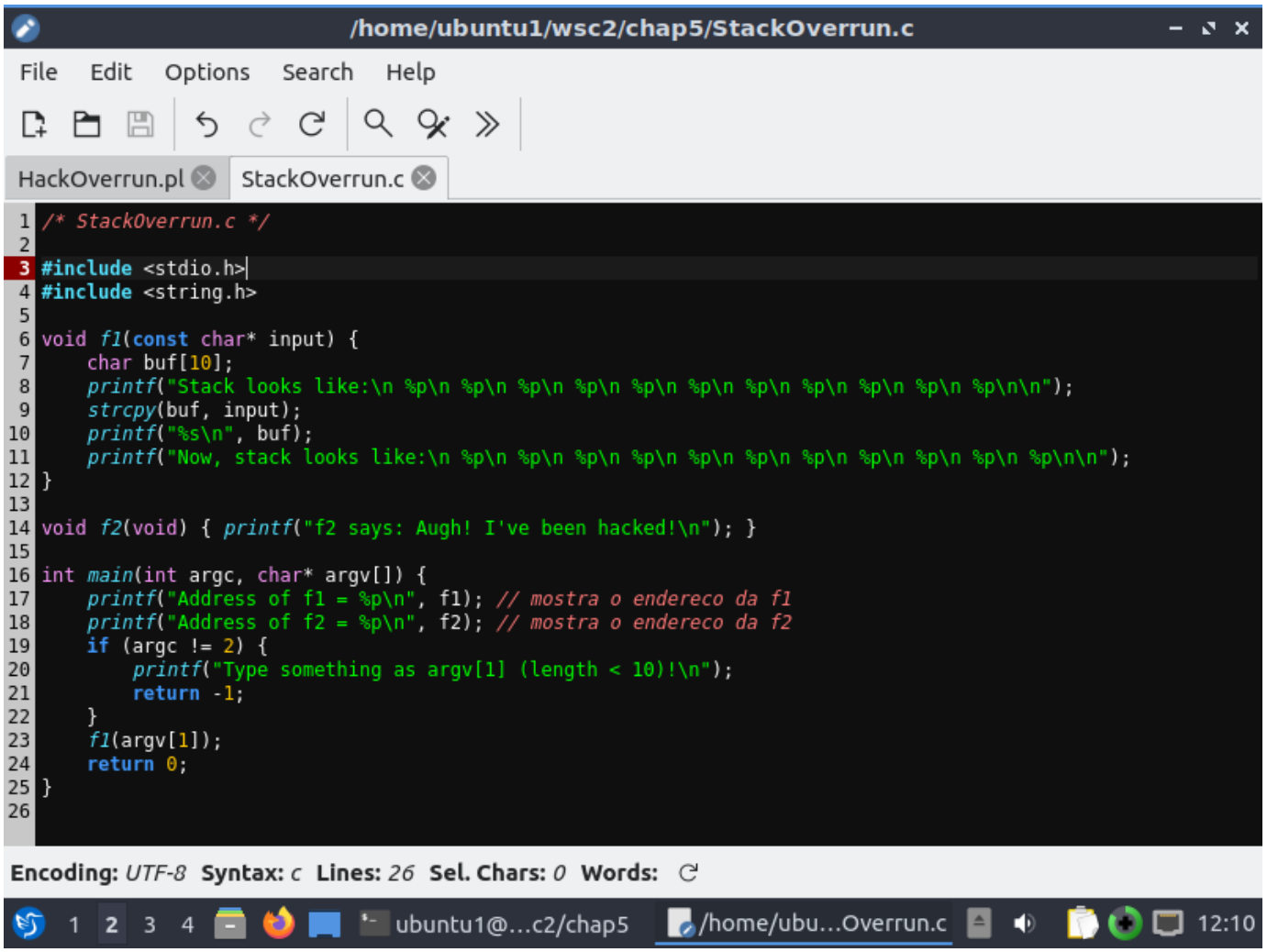


OpenSSL's HeartBleed (April, 2014)

- Presente no TLS Heartbeat do OpenSSL
- Causado pela falta de verificação de limites de um buffer contendo um nonce
- Em C, o tamanho do buffer e a variável que o contém são desassociados
- Poderia ser identificado por revisão de código
- Poderia ser evitado por tipos de dados fortes
- Ainda, pouca diversidade de boa criptografia
- Não são intercambiáveis



O programa StackOverrun.c



- O programa demonstra o conceito de buffer overflow sobre a stack de um programa em C.
- Programa adaptado de StackOverrun.c, capítulo 5 do livro Writing Secure Code, 2nd Ed.
- O arquivo com os programas de exemplo pode ser obtido (legal e free) no link https://resources.oreilly.com/examples/9780735617223-files

O programa StackOverrun.c

A função f1() faz o seguinte:

- imprime a stack do programa antes do buffer buf;
- copia com strcpy() o input para o buffer buf;
- imprime novamente a stack.

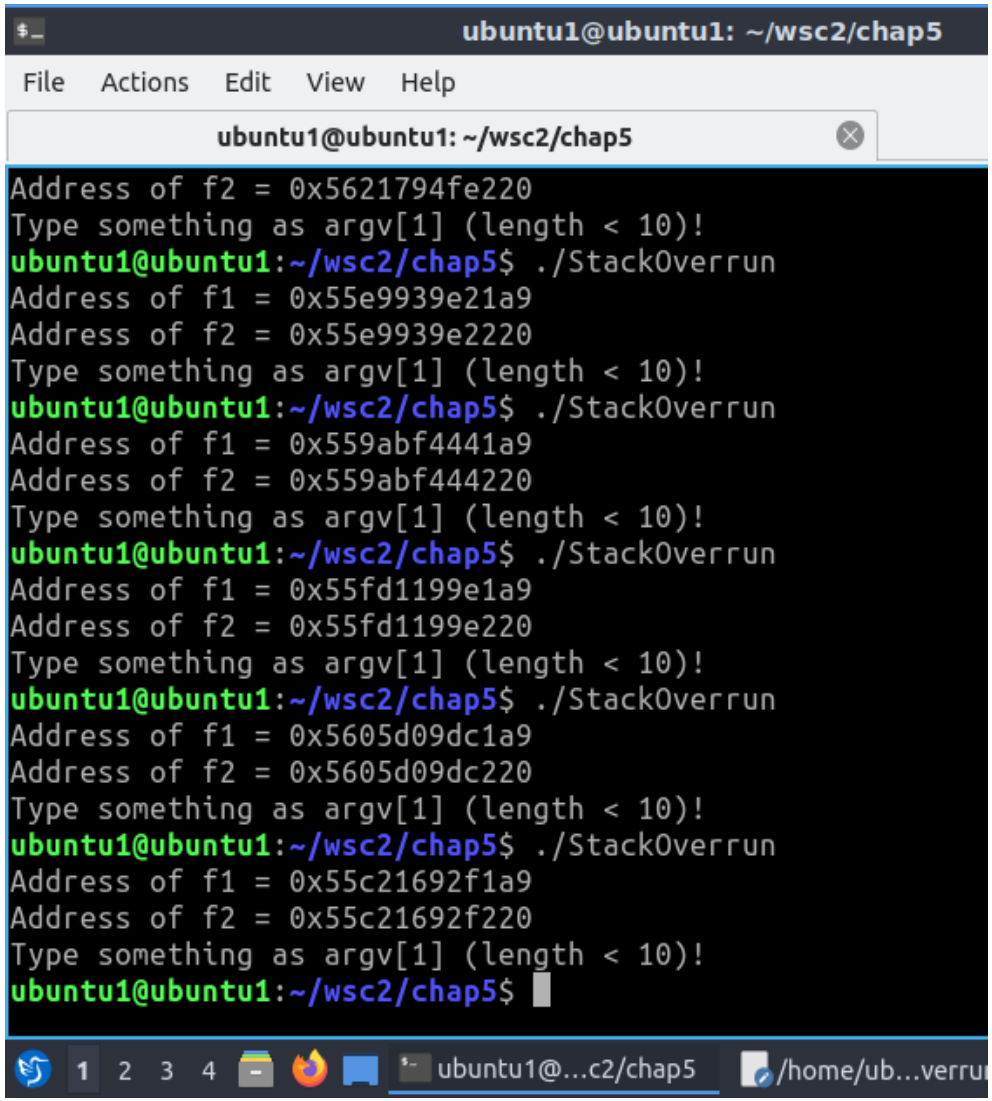
A função f2() somente imprime uma mensagem de aviso/alerta.

A função main() faz o seguinte:

- imprime o endereço da função f1();
- imprime o endereço da função f2();
- verifica se algum parâmetro foi passado;
- se não foi, avisa para passar um parâmetro e sai do programa;
- chama a função f1() com a parâmetro arg[1].

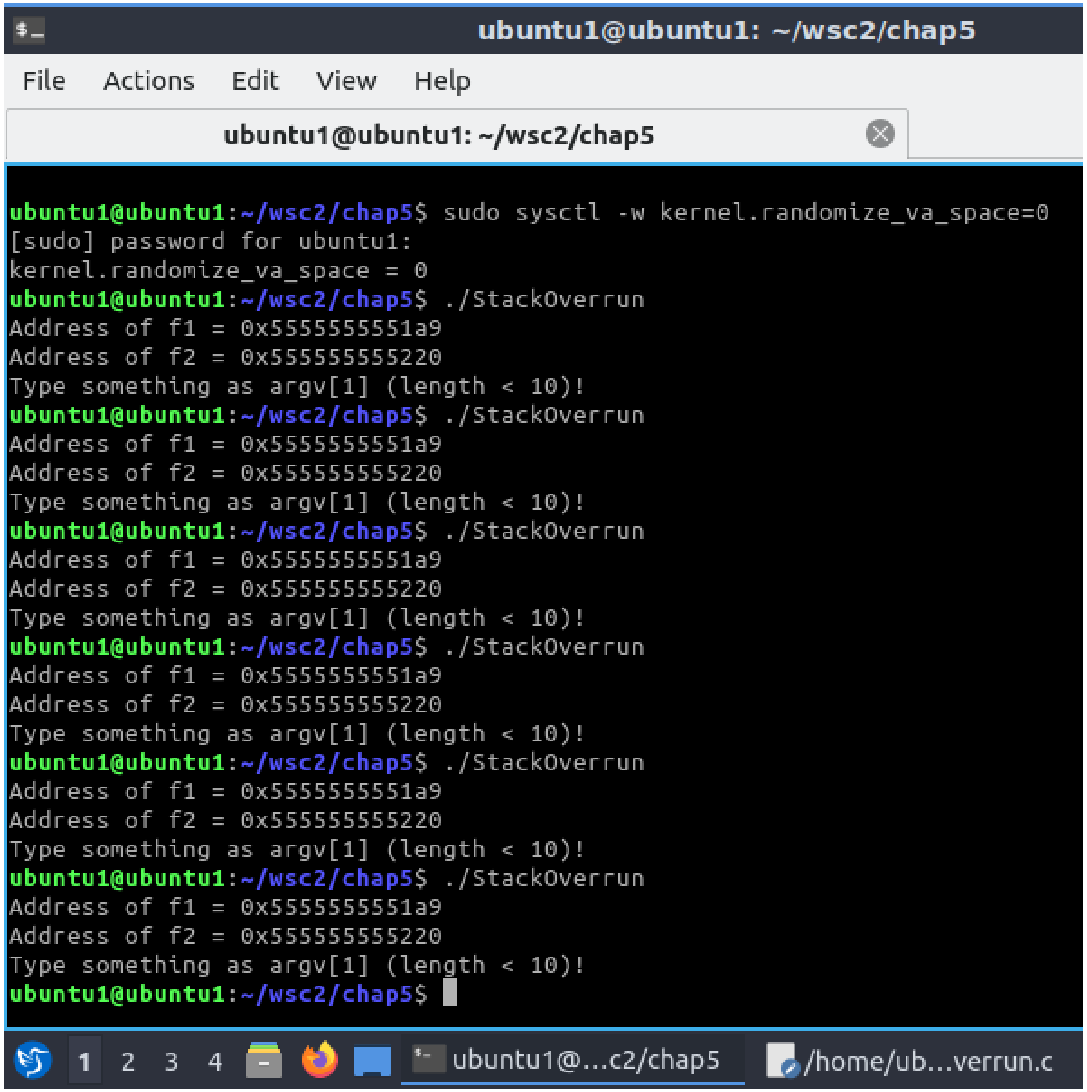
O objetivo da demonstração de exploração é executar a função f2() mesmo que ela não tenha sido chamada explicitamente no main().

Compilando e executando StackOverrun.c



- Em uma máquina Ubuntu Linux (real ou virtual) com gcc instalado.
- Compilando o programa sem restrições e avisos (warnings) desligados.
  - gcc -w -o StackOverrun StackOverrun.c
- Executar o programa StackOverrrun várias vezes sem parâmetros
  - ./StackOverrun
  - O que acontece?
- Observar o endereço das funções f1() e f2() é diferente

Compilando e executando StackOverrun.c



Sistemas operacionais modernos implementam randomização de endereços virtuais no kernel. A aleatorização dos endereços virtuais atrapalha a exploração do overflow, uma vez que é uma dificuldade extra para atingirmos o endereço de f2() a partir da stack.

Os comandos alteram a configuração para randomização de endereços na VM da demonstração.

Desliga a randomização de endereços

- `sudo sysctl -w kernel.randomize_va_space=0`

•Restaura a randomização de endereços

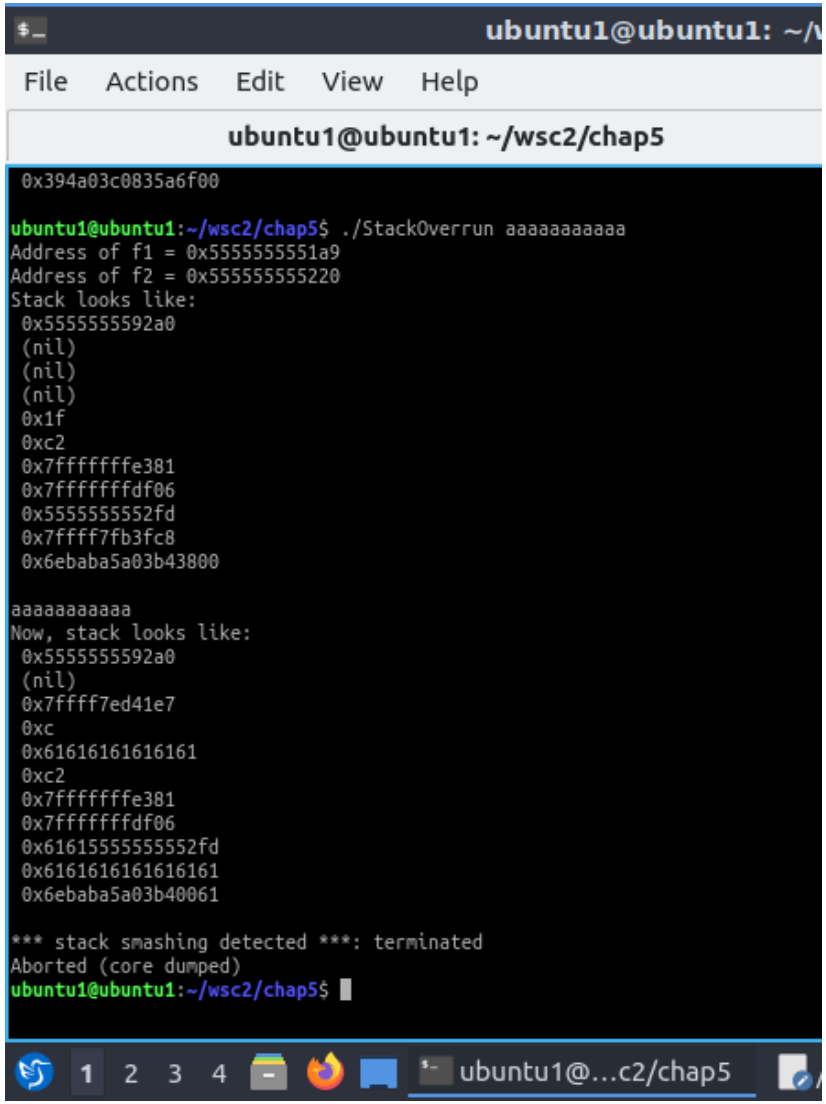
- `sudo sysctl -w kernel.randomize_va_space=2`

Desligar a aleatorização de endereços e executar novamente o programa.

- `sudo sysctl -w kernel.randomize_va_space=0`
- `./StackOverrun`

Desta vez, os endereços de `f1()` e `f2()` se repetem em todas as execuções!

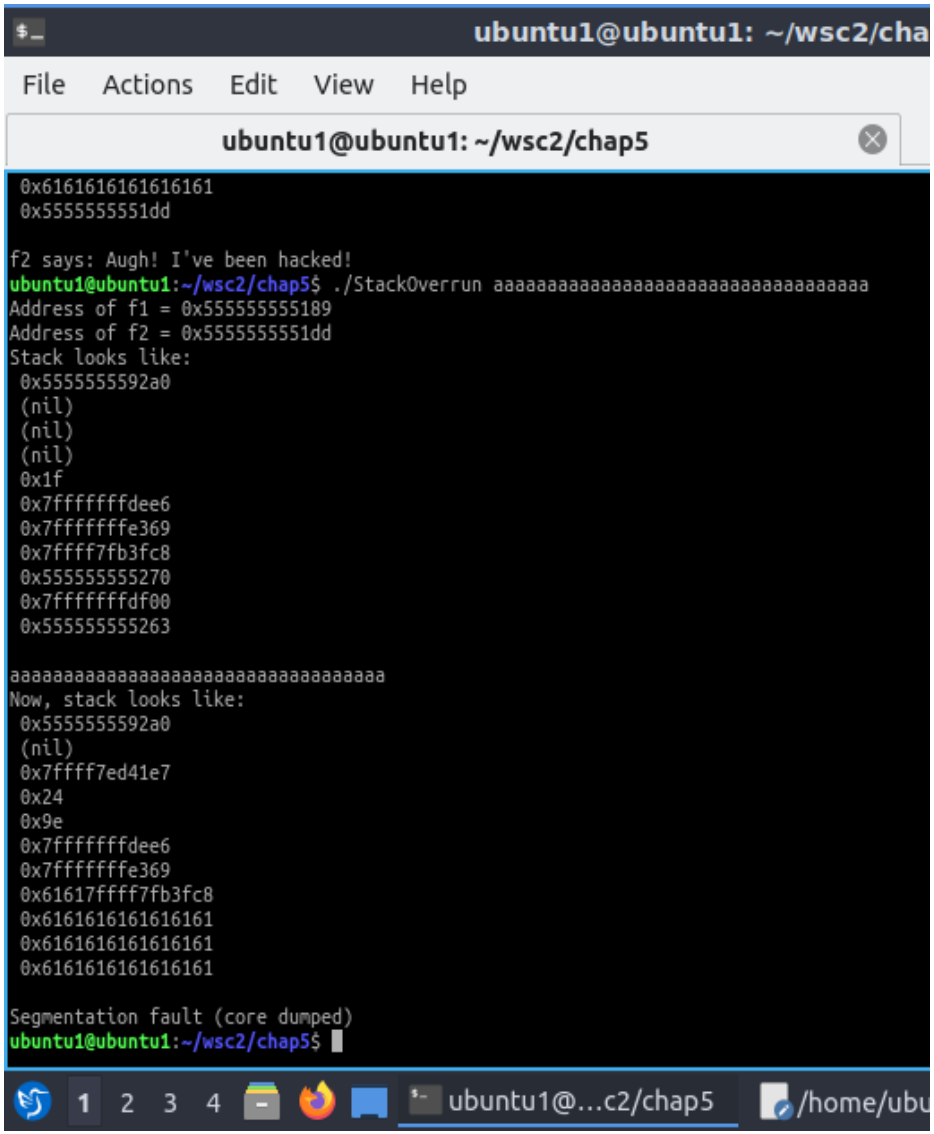
Compilando e executando StackOverrun.c



- Executar o programa com os parâmetros `aaaaa`
  - `./StackOverrun aaaaa`
  - Esta execução termina o programa de modo bem sucedido

- Executar o programa com os parâmetros
  - `./StackOverrun aaaaaaaaaaa`
  - Esta execução termina o programa de modo anormal
  - Além do `core dump`, algo diferente aconteceu
    - A última linha da stack é o endereço de retorno e foi corrompido!
    - A adulteração da stack foi detectada por quem?

Compilando e executando StackOverrun.c



Os compiladores modernos permitem a detecção de adulteração da stack e também o uso de stack não executável (se a stack não é executável, código malicioso injetado nela não é executado).

O comando gcc a seguir compila o programa StackOverrun sem as proteções de stack incluídas pelo compilador e com a stack executável.

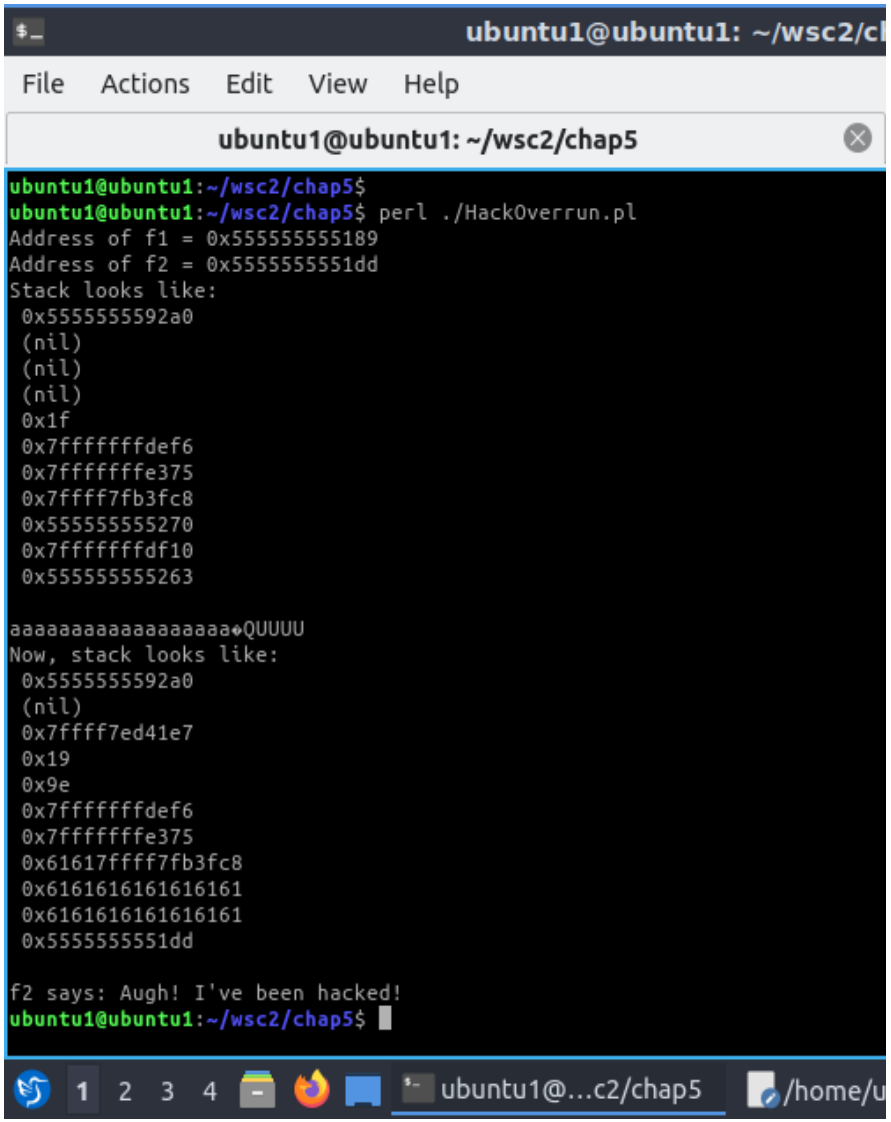
- `gcc -w -fno-stack-protector -z execstack -o StackOverrun StackOverrun.c`

- Uma curiosidade: o código executável fica menor sem os controles incluídos pelo compilador.

- Executar o programa com os parâmetros
  - `./StackOverrun aaaaaaaaaaa`
  - Esta execução termina o programa de modo anormal
  - A adulteração da stack NÃO foi detectada!

Agora é possível manipular a pilha para alterar o endereço de retorno. Para tal, é necessário passar para o programa o novo endereço de retorno no parâmetro arg[1] e codificado em hexadecimal.

Explorando a vulnerabilidade Stack Overflow



O script perl `HackOverrun.pl` pode ser usado para codificar o novo endereço de retorno.

- O endereço de f2() será usado aqui.
  - `0x5555555551dd`

HackOverrun.pl | StackOverrun.c

```
1
2 $arg = "aaaaaaaaaaaaaaaaaa"."\xdd\x51\x55\x55\x55\x55";
3 $cmd = "./StackOverrun ".$arg;
4
5 system($cmd);
6
7
```

O script concatena duas strings

- `"aaaaaaaaaaaaaaaaaa"`
  - com 18 'a' consecutivos
- `"\xdd\x51\x55\x55\x55\55"`
  - a parte específica, menos significativa, do endereço de `f2()`

O script chama o programa `StackOverrun` com o parâmetro malicioso.

O resultado da execução do script é mostrado na figura seguir.

- `perl ./HackOverrun.pl`

f2() é executada mesmo sem ser explicitamente chamada no código fonte do programa principal!!!

- A mensagem é emitida: "`f2 says: Argh! I've been hacked!`"

Soluções: String Libraries seguras

- C – evitar funções sem bounds check:
  - strcpy(), strcat(), sprintf(), scanf()
- Usar versões seguras (com bound check):
  - strncpy(), strncat(), fgets()

- Microsoft's *StrSafe*, Messier&Viega's *SafeStr* fazem bound check e terminação com null
  - É preciso passar o tamanho certo do buffer para a função
- C++: STL string class trata alocação com segurança

- Diferente das linguagens compiladas (C/C++), as interpretadas (Java/C#) reforçam *type safety*, e levantam exceções para *buffer overflow*

Soluções: Outras soluções

- Reescrita de código antigo de manipulação de strings
  - Solução geralmente cara
- StackGuard (canários)
- Análise estática
- Non-executable stacks
- Linguagens interpretadas com sandbox (Java, C#)

Resumo sobre buffer overflow

- Buffer overflow tem sido por muito tempo uma das ameaças mais comuns!
  - Usada em vários malwares (Morris Worm)
  - Afeta tanto stacks quanto heaps
- Atacante pode executar o código que quiser, sequestrar programas
- Prevenção pela verificação de limites de todos os buffers
  - e/ou canários e análise estática
- É um tipo de corrupção de memória, há outros:
  - Format String, Integer Overflow, etc…
- Ameaça constante em sistemas embarcados e/ou restritos
  - p.ex. IoT, PoS

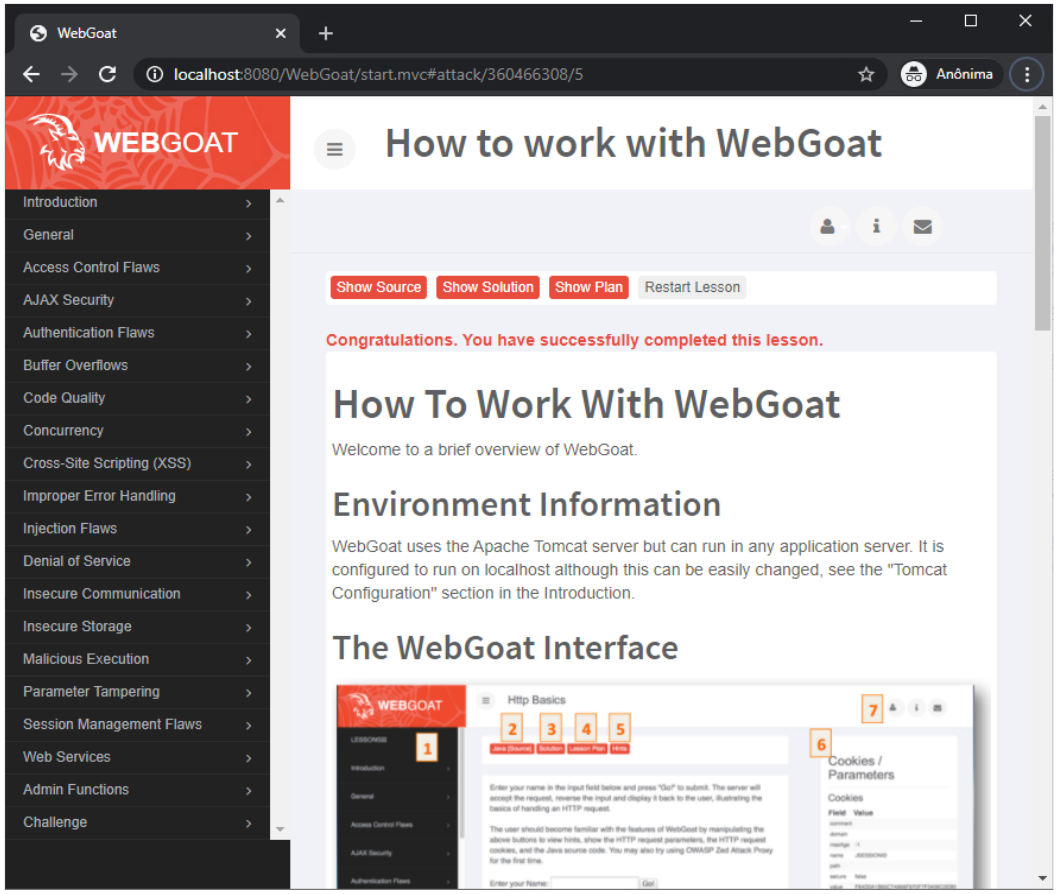
Parte 3-2 (Laboratório)

bWAPP - buggy Web APPlication

- Os tutoriais da bWAPP podem ser facilmente achados na Internet
- Tutorial oficial: https://www.mmebvba.com/sites/default/files/downloads/bWAPP_intro.pdf
- Outro tutorial interessante: https://wooly6bear.files.wordpress.com/2016/01/bwapp-tutorial.pdf
- Uma VM standalone do bWAPP: https://sourceforge.net/projects/bwapp/files/bee-box/
- Logar na aplicação bWAPP com login “bee” e senha “bug”.

OWASP WebGoat

- A página do WebGoat está no link https://owasp.org/www-project-webgoat/
- WebGoat no github https://github.com/WebGoat/WebGoat
- https://github.com/WebGoat/WebGoat/releases
- Versão 7.1 https://github.com/WebGoat/WebGoat/releases/tag/7.1
- Para executar a versão 7.1 com Java 8+:
- `java -jar webgoat-container-7.1-exec.jar`
- No Browser: http://localhost:8080/WebGoat
- Logar na aplicação com o usuário guest/guest



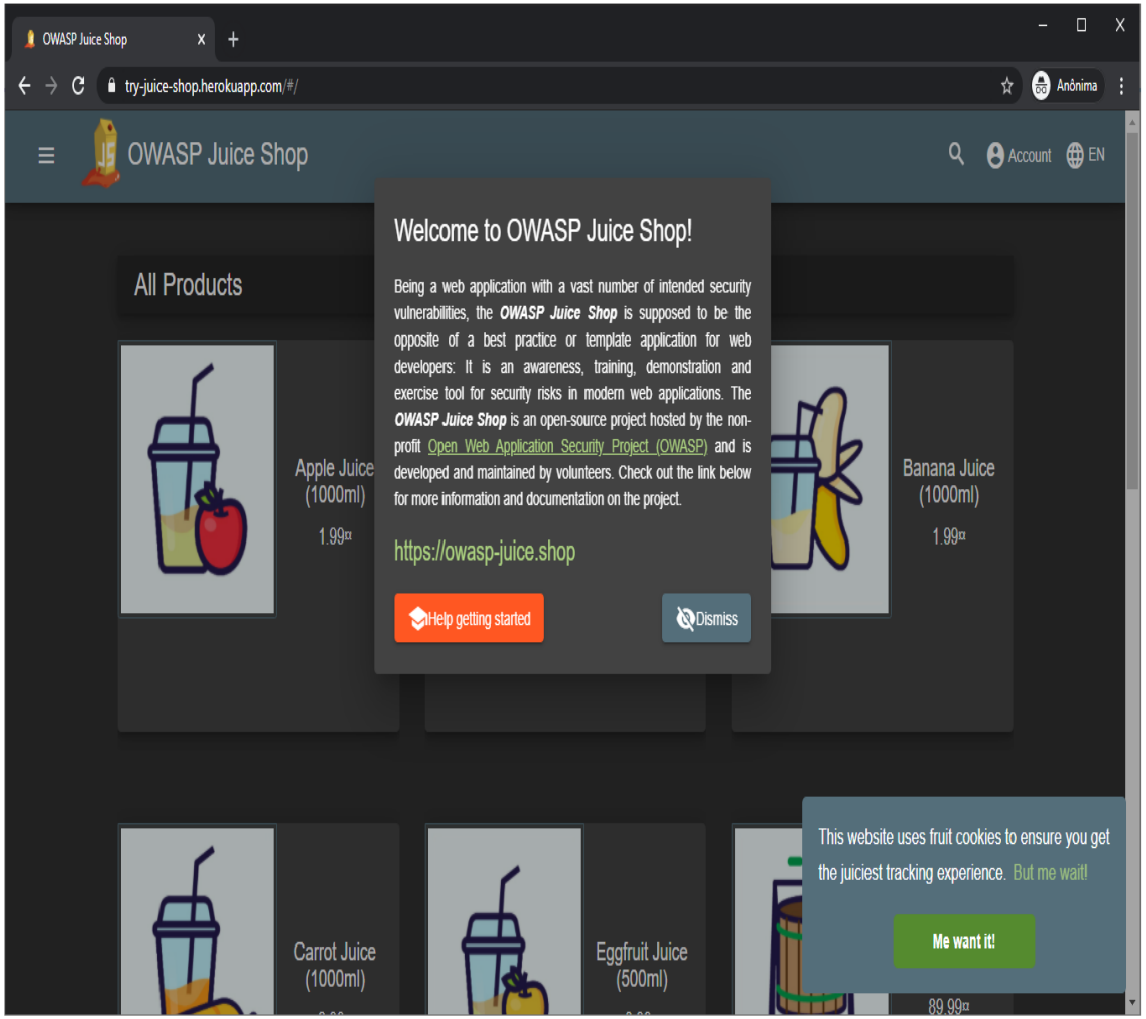
OWASP Juice Shop

- Página OWASP Juice Shop
- https://owasp.org/www-project-juice-shop

- Opções de instalação e/ou implantação
- https://github.com/bkimminich/juice-shop#setup

- Instalação do instrutor na heroku.com
- https://try-juice-shop.herokuapp.com

- Um guia de testes
- https://pwning.owasp-juice.shop



Obrigado!