Web Application Security

Lecture:04

Bypassing Client-Side Controls

Miss Maryam Malik
Lecturer
Department of cyber security
Air university Islamabad

Clients Repeat Data

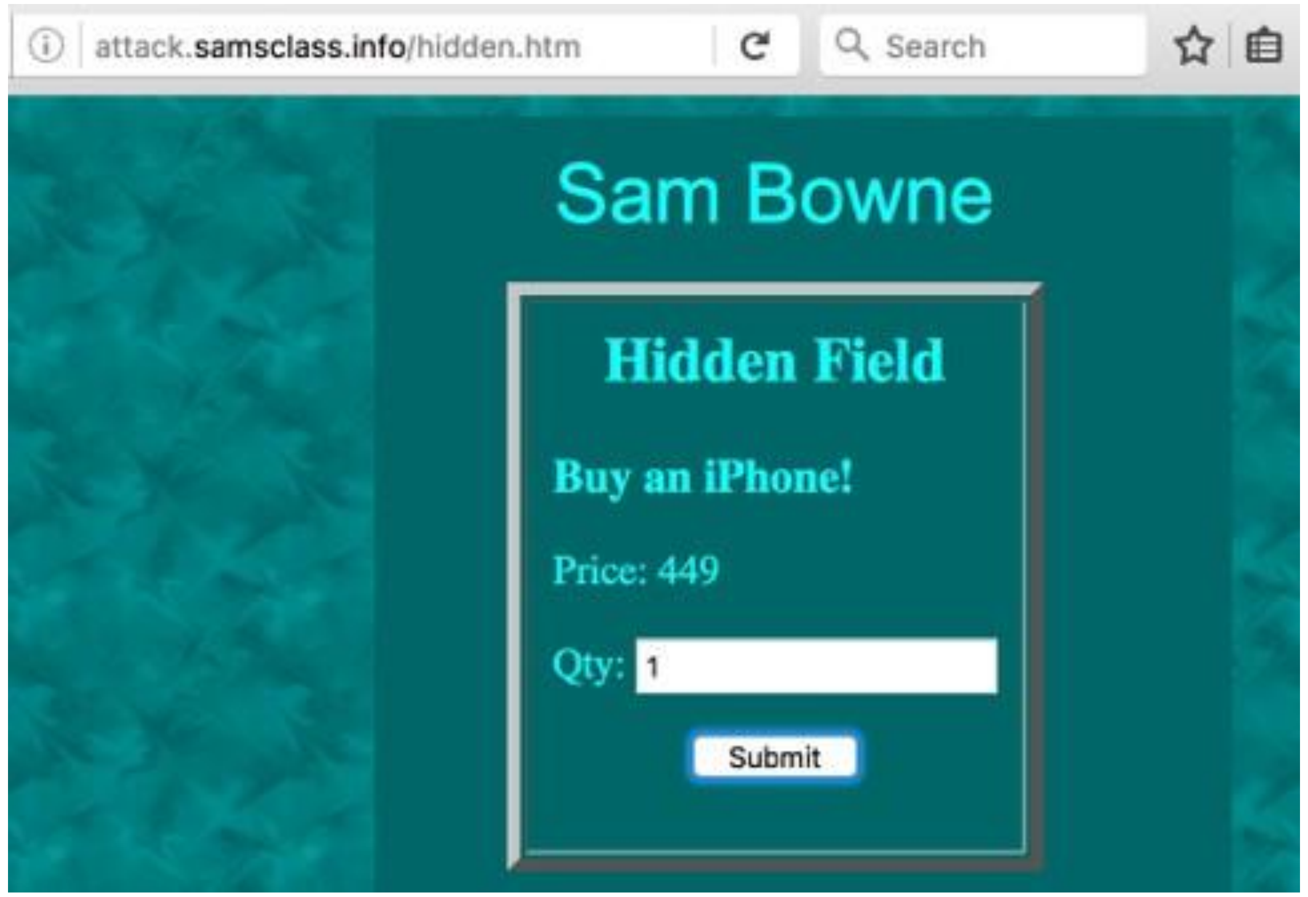
- **It's common for a server to send data to a client**
- **And for the client to repeat that same data back to the server**
- **Developers often assume that the client won't modify the data**

Why Repeat Data?

1. **Less data on the server**: By sending data to the client, developers don’t have to store as much information on the server, which can make the app faster and easier to manage.
2. **Multiple servers**: If the application uses several servers and the user interacts with more than one, it can be hard to share data between those servers. Sending data via the client can be a quick fix.
3. **Third-party components**: If the application uses third-party tools (like shopping carts), developers might find it hard to modify them. Sending data to the client may be an easier workaround.
4. **Avoiding bigger changes**: Storing new data on the server might require big updates or testing. Sending data via the client is a faster, simpler solution when time is tight.

Hidden Form Fields

Server sends hidden price field to client



attack.samsclass.info/hidden.htm

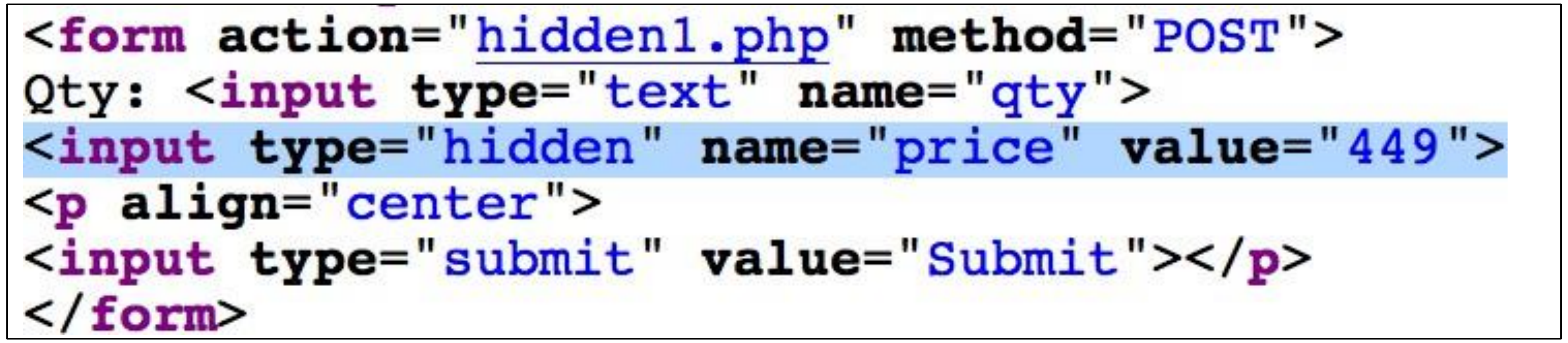
Sam Bowne

Hidden Field

Buy an iPhone!

Price: 449

Qty: 1

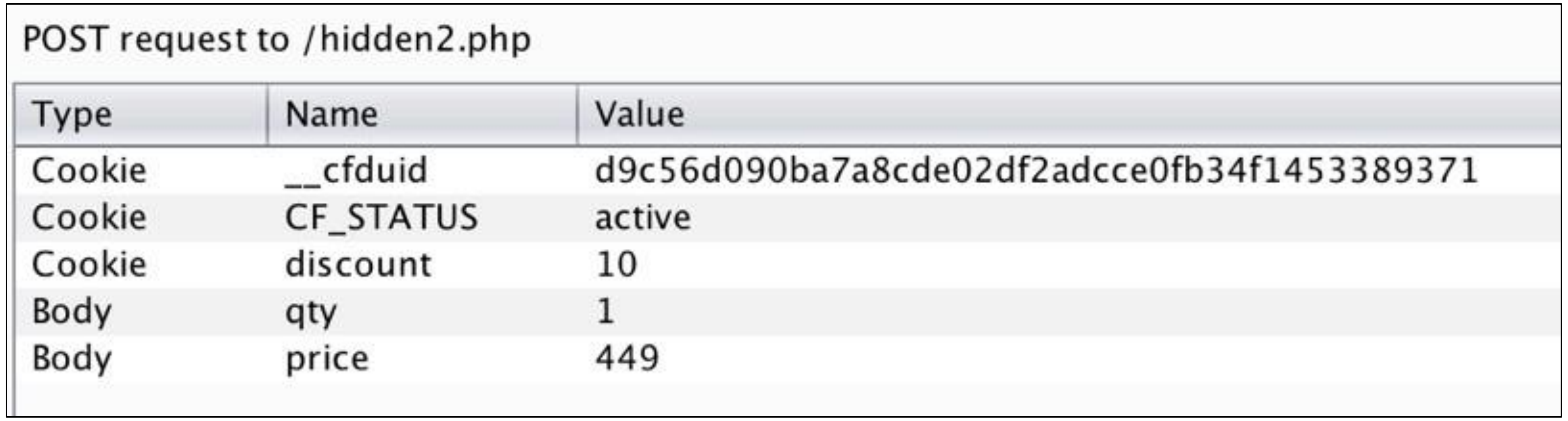


```
<form action="hidden1.php" method="POST">
Qty: <input type="text" name="qty">
<input type="hidden" name="price" value="449">
<p align="center">
<input type="submit" value="Submit"></p>
</form>
```

Cookie Fields

Discount amount in cookie

```
HTTP/1.1 200 OK
Set-Cookie: discount=10
Content-Length: 1530
```

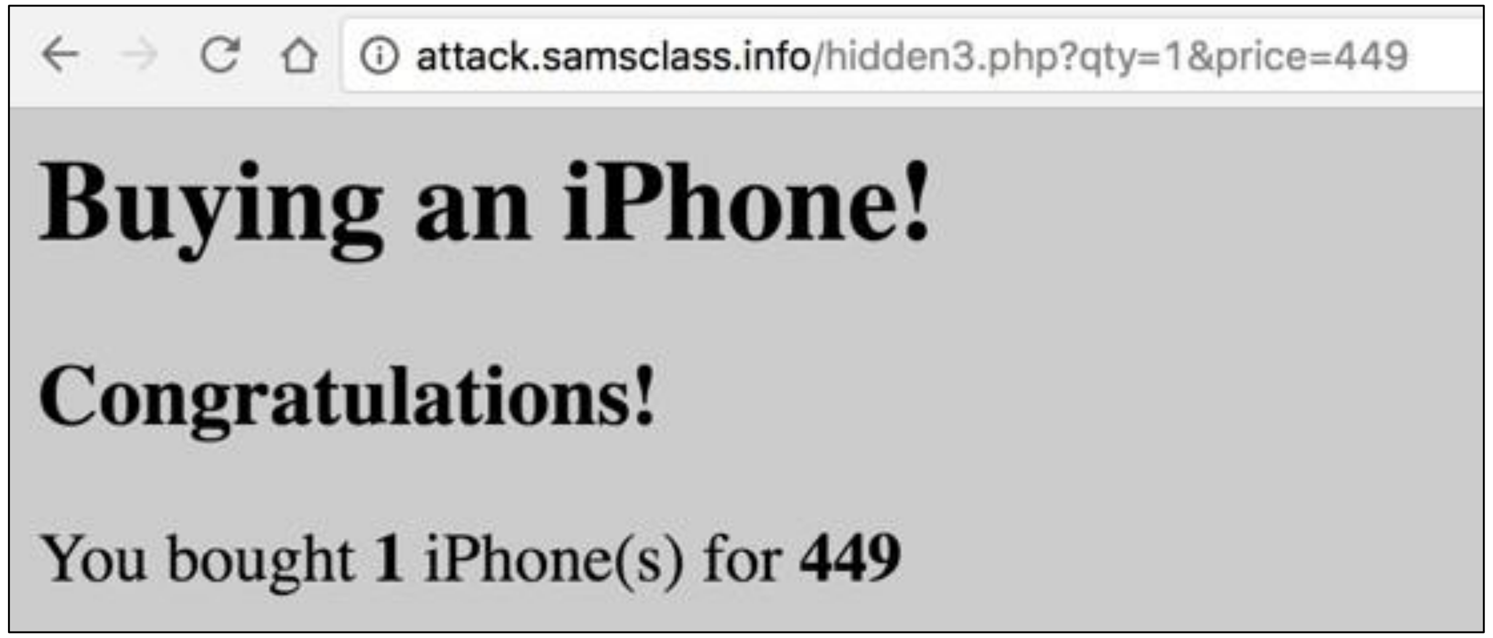


POST request to /hidden2.php

Type	Name	Value
Cookie	__cfduid	d9c56d090ba7a8cde02df2adcce0fb34f1453389371
Cookie	CF_STATUS	active
Cookie	discount	10
Body	qty	1
Body	price	449

URL Parameters

These parameters appear in the URL (the web address) after a question mark (?). For example, **http://mdsec.net/shop/?prod=3&pricecode=32**

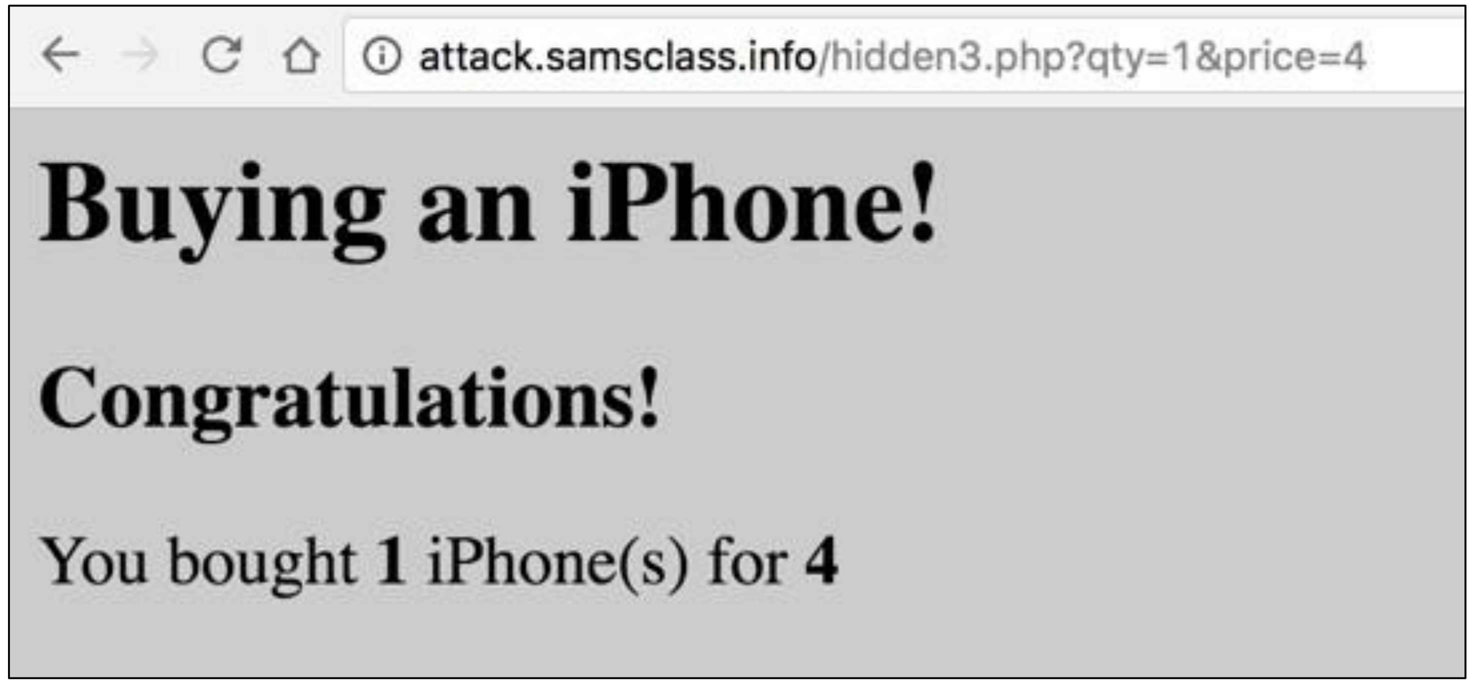


attack.samsclass.info/hidden3.php?qty=1&price=449

Buying an iPhone!

Congratulations!

You bought **1** iPhone(s) for **449**



attack.samsclass.info/hidden3.php?qty=1&price=4

Buying an iPhone!

Congratulations!

You bought **1** iPhone(s) for **4**

No proxy needed

Just modify the URL

Hidden URL Parameters

```
<img src="http://foo.com?price=449">

<iframe src="http://foo.com?price=449">

<form action="http://foo.com?price=449"
method="POST">
```

- **All are unsafe; can be exploited with a proxy**

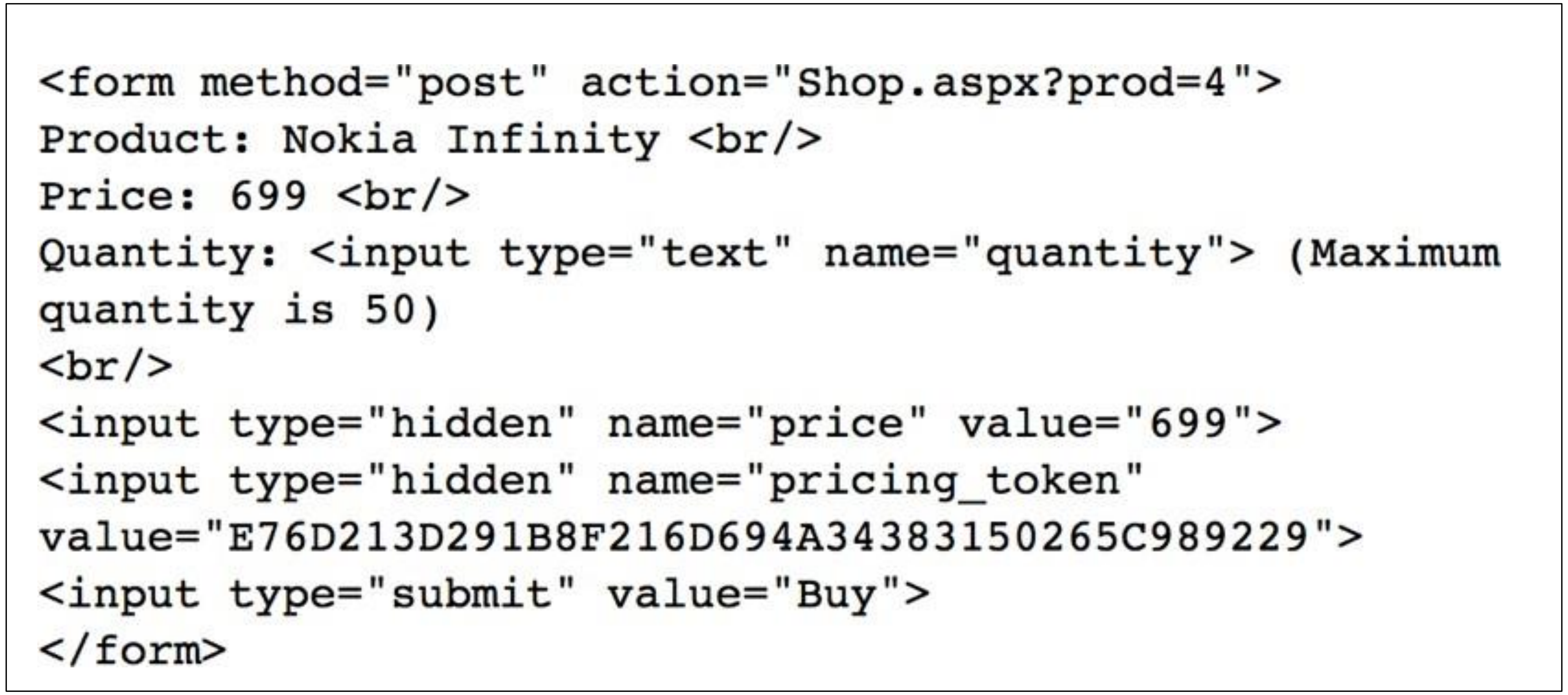
Referer Header

```
GET /auth/472/CreateUser.ashx HTTP/1.1
Host: mdsec.net
Referer: https://mdsec.net/auth/472/Admin.ashx
```

- **Shows the URL that sent the request**
- **Developers may use it as a security mechanism, trusting it**

Opaque Data

- **Data may be encrypted or obfuscated**



```
<form method="post" action="Shop.aspx?prod=4">
Product: Nokia Infinity <br/>
Price: 699 <br/>
Quantity: <input type="text" name="quantity"> (Maximum
quantity is 50)
<br/>
<input type="hidden" name="price" value="699">
<input type="hidden" name="pricing_token"
value="E76D213D291B8F216D694A34383150265C989229">
<input type="submit" value="Buy">
</form>
```

Handling Opaque Data

- **If you know the plaintext, you may be able to deduce the obfuscation algorithm**
- **App may contain functions elsewhere that you can leverage to obfuscate plaintext you control**
- **You can replay opaque text without deciphering it**
- **Attack server-side logic with malformed strings, such as overlong values, different character sets, etc.**

ASP.NET ViewState

Loading the Page:

Imagine you’re on a shopping website looking at a phone called HTC Avalanche. The price is $399. The website saves this price in the hidden __VIEWSTATE so it can remember it later.

```
<form method="post" action="Shop.aspx?prod=3">
   <input type="hidden" name="__VIEWSTATE" id="__VIEWSTATE"
value="someEncodedValue"/>
   Product: HTC Avalanche <br/>
   Price: 399 <br/>
   Quantity: <input type="text" name="quantity"> (Maximum quantity is 50)
   <br/>
   <input type="submit" value="Buy">
</form>
```

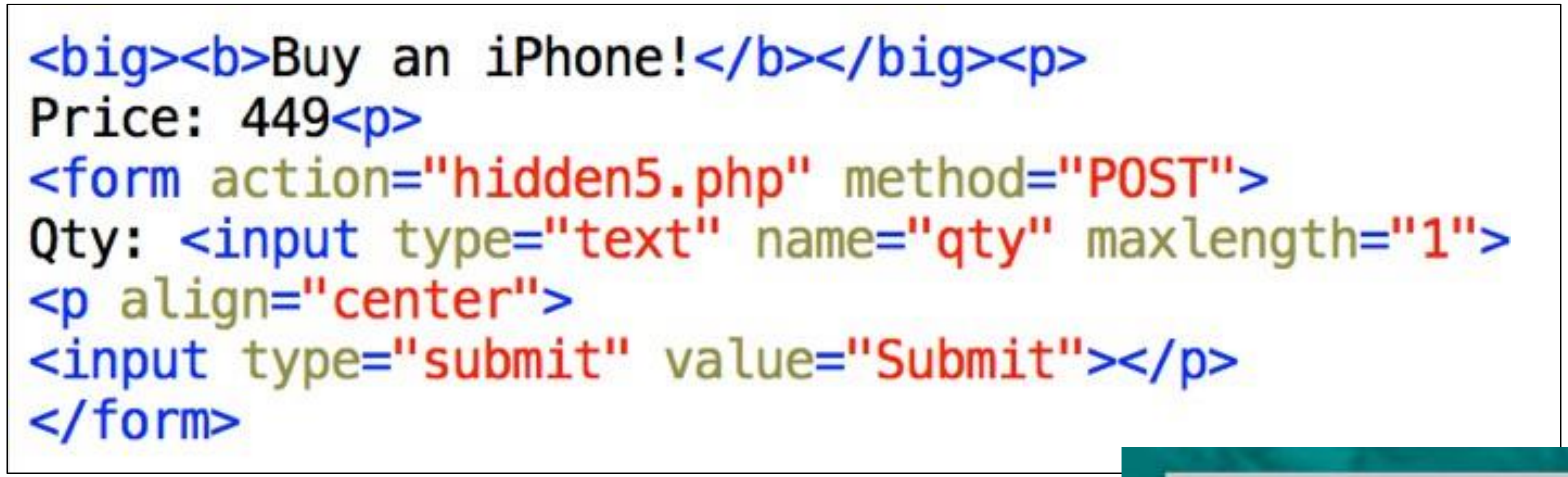
ViewState

Submitting the Form:
When you type in how many of the phone you want (let's say 1) and click the "Buy" button, the website sends both your quantity and the hidden ViewState back to the server.

```
POST /shop/76/Shop.aspx?prod=3 HTTP/1.1
__VIEWSTATE=someEncodedValue&
quantity=1
```

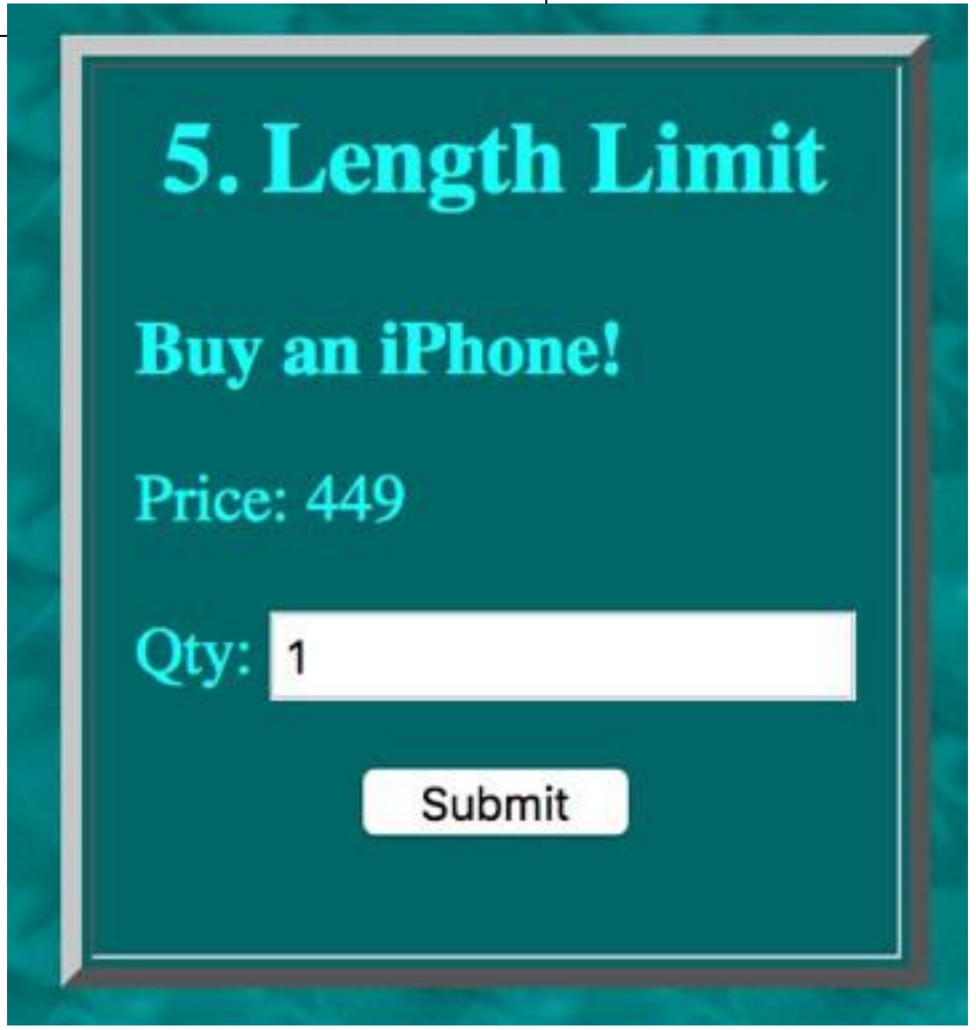
The server looks at the ViewState to see the price of the phone.
If the ViewState is changed in a way that the server doesn’t expect, it will give an error. That’s because it relies on the information in the ViewState to function correctly.

HTML Forms



```
<big><b>Buy an iPhone!</b></big><p>
Price: 449<p>
<form action="hidden5.php" method="POST">
Qty: <input type="text" name="qty" maxlength="1">
<p align="center">
<input type="submit" value="Submit"></p>
</form>
```

- **Only allows one character**
- **Intending to set max. of 9**



5. Length Limit

Buy an iPhone!

Price: 449

Qty: 1

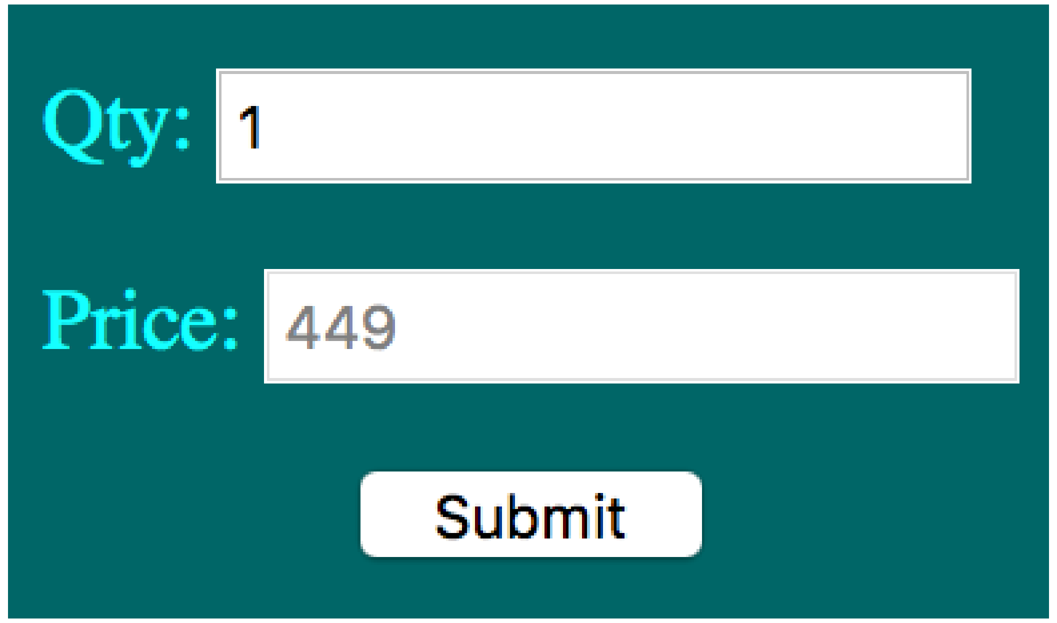
Script-Based Validation

Script-based validation uses JavaScript to check if user input is valid before sending the form to the server.

```
<form method=”post” action=”Shop.aspx?prod=2” onsubmit=”return
validateForm(this)”>
Product: Samsung Multiverse <br/>
Price: 399 <br/>
Quantity: <input type=”text” name=”quantity”> (Maximum quantity is 50)
<br/>
<input type=”submit” value=”Buy”>
</form>
<script>function validateForm(theForm)
{
var isInteger = /^\d+$/;
var quantity = theForm.quantity.value;
var valid = isInteger.test(quantity) && quantity > 0 && quantity <= 50;
if (!valid)
alert(’Please enter a valid quantity’);
return valid;
}
</script>
```

Disabled Elements

```
<form action="hidden7.php" method="POST">
Qty: <input type="text" name="qty"><p>
Price: <input type="text" disabled="true"
name="price" value="449">
<p align="center">
<input type="submit" value="Submit"></p>
</form>
```

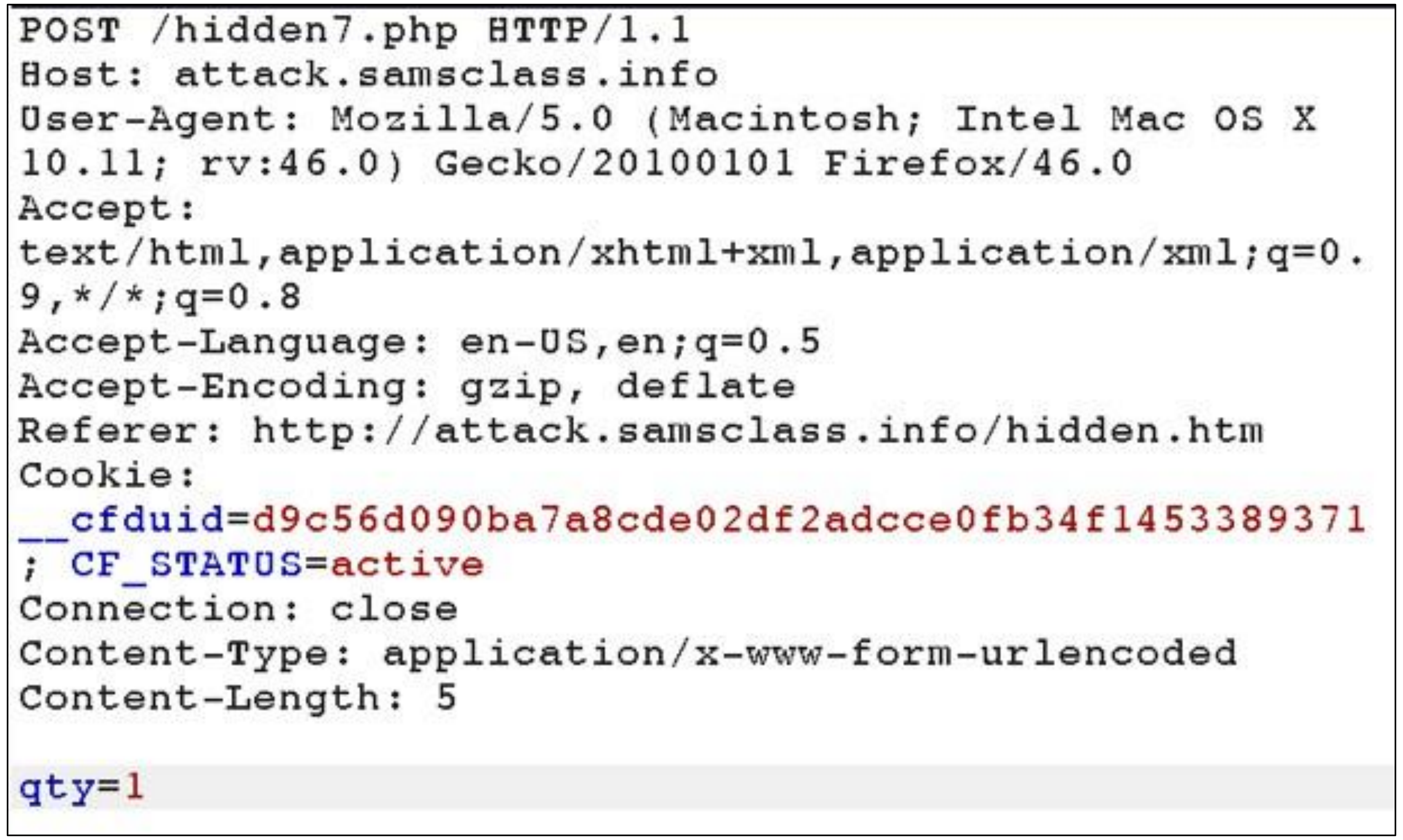


Qty: 1

Price: 449

Disabled Elements

- **Cannot be changed**
- **Not sent to server**



```
POST /hidden7.php HTTP/1.1
Host: attack.samsclass.info
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X
10.11; rv:46.0) Gecko/20100101 Firefox/46.0
Accept:
text/html,application/xhtml+xml,application/xml;q=0.
9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://attack.samsclass.info/hidden.htm
Cookie:
__cfduid=d9c56d090ba7a8cde02df2adcce0fb34f1453389371
; CF_STATUS=active
Connection: close
Content-Type: application/x-www-form-urlencoded
Content-Length: 5

qty=1
```

Browser Extensions

- **Flash or Java client-side routines can collect and process user input**
  - **Internal workings are less transparent than HTML forms and JavaScript**
  - **But still subject to user modification**

Example: Casino App

- **Client could**
  - **Tamper with game state to gain an advantage**
  - **Bypass client-side controls to perform illegal actions**
  - **Find a hidden function, parameter, or resource to gain illegitimate access to a server-side resource**
  - **Receive information about other players to gain an advantage**

Common Browser Extensions

- **Java applets, Flash, and Silverlight**
- **All have these features**
  - **Compiled to bytecode**
  - **Execute in a virtual machine that provides a sandbox**
  - **May use remoting frameworks employing serialization to transmit complex data structures or objects over HTTP**

Java

Java applets are small programs that run in the **Java Virtual Machine (JVM)**, a special environment within your browser.

These applets are designed to be isolated from the rest of your computer by the **Java Security Policy**, which provides **sandboxing**—a way to restrict what the applet can access or modify on the user's system.

Flash

Flash Virtual Machine: Flash runs in its own environment called the Flash virtual machine, which keeps it separate from the user’s computer system, similar to how Java applets run in the JVM.
Sandboxing: Flash objects are also sandboxed, meaning they have restricted access to the host system. This helps prevent them from causing harm or accessing sensitive data on the user’s computer.
ActionScript: This is the programming language used to create Flash applications. The latest version, ActionScript 3, introduces enhanced capabilities, allowing developers to build more sophisticated applications.

Silverlight

Silverlight is a web application framework developed by Microsoft, designed to create rich internet applications with features similar to Flash.

Rich Internet Applications: Silverlight enables developers to build applications that provide a rich user experience, akin to desktop applications, directly in the web browser.
Sandboxed Environment: Like Java and Flash, Silverlight applications run in a sandboxed environment. This means they have restricted access to the user's computer resources, enhancing security and preventing potential harm.
.NET Framework Integration: Silverlight applications can be developed using any .NET-compliant language, with C# being the most commonly used. This allows developers familiar with .NET technologies to create interactive web applications.
Cross-Browser Compatibility: Silverlight was designed to run on various web browsers and operating systems, although it requires a browser plugin.

Approaches to Browser Extensions

When testing applications that use browser extensions, you can employ two main techniques: **intercepting and modifying requests/responses** and **analyzing the component directly**.

Intercepting and Modifying Requests/Responses: This technique involves using tools (like intercepting proxies) to capture and analyze the HTTP requests sent by the browser extension to the server and the responses received.

Limitations

1. Obfuscation and Encryption
2. Missing Functionality
3. Configuration Challenges

Analyzing the Component Directly: This technique involves decompiling the browser extension’s bytecode to view the original source code or using a debugger to interact with the component dynamically.

Limitation

Time-Consuming: This method can take a significant amount of time and effort, especially if you're not familiar with the technology or programming languages used.
Technical Knowledge Required: A thorough understanding of the extension’s framework and coding is necessary, which might be a barrier for some testers

Intercepting Traffic from Browser Extensions

Sometimes, testing can become difficult due to obstacles like:

1. **Encrypted or Obfuscated Data**: The data transmitted by the extension may be encrypted, serialized, or obfuscated, making it difficult to understand or modify.

2. **Custom Protocols or Serialization Formats**: The browser extension might use a custom protocol or data format, which requires specific knowledge or tools to intercept and modify effectively.

Java Serialization

Web applications sometimes serialize data or objects before sending them in HTTP requests.

Example: Java Serialization
Java applets often use **native serialization** to send objects between client and server.

```
Content-Type: application/x-java-serialized-object
```

•**Intercepting**: Use a proxy to capture the serialized data.
•**Deserialization**: Use tools like **DSer**, a Burp Suite plugin, to convert the serialized data
• into an editable format, such as XML.
•**Editing**: Modify the data within the deserialized structure.
•**Reserialization**: DSer then helps reserialize the modified data and update the
• HTTP request accordingly.

Flash Serialization

Flash uses a serialization format called Action Message Format (AMF) to transmit complex data structures between client and server components.

When you see a Content-Type:

```
Content-Type: application/x-amf
```

header in an HTTP request or response, it indicates that the data is serialized in AMF format.

Silverlight Serialization

Silverlight applications often use Windows Communication Foundation (WCF), a remoting framework built into the .NET platform, to communicate between client and server components.
They typically use a serialization format known as Microsoft’s .NET Binary Format for SOAP (NBFS).

You can identify serialized NBFS data in an HTTP request or response by looking for the Content-Type:

```
Content-Type: application/soap+msbin1
```

Obstacles to Intercepting Traffic from Browser Extensions

1.Proxy Configuration Issue: Some browser extensions may bypass your browser’s proxy settings and make their own HTTP requests. To intercept such requests, you can modify your computer’s hosts file and configure your proxy for invisible proxying, allowing automatic redirection.
2 .SSL Certificate Issue: Extensions may reject the SSL certificate used by your intercepting proxy. To resolve this, you can set up your proxy to use a master CA certificate that signs individual certificates for each site, then install that CA certificate in your trusted certificate store.

If the extension uses a **non-HTTP protocol**, you can use a tool like **Echo Mirage** to intercept and modify network data by injecting into the extension's process and monitoring its socket API calls.

Decompiling Browser Extensions

decompiling Browser Extensions refers to the process of reversing the bytecode of browser extension components back into readable source code. This method gives attackers (or security testers) the most in-depth understanding of how a browser extension works and any vulnerabilities it may contain. Let’s break this down step by step.

Why Decompile?

When a browser extension is used in a web application, it’s typically compiled into **bytecode** (a platform-independent binary format). Decompiling the bytecode allows you to:

1. **Review the Source Code**: By reversing bytecode into its original code (or something very close), you can analyze how the extension works, find vulnerabilities, or understand its logic.
2. **Modify the Behavior**: Once you have decompiled the code, you can make changes to how the extension behaves. Afterward, you can recompile it with your modifications and run it again.
3. **Identify Security Flaws**: Many vulnerabilities that aren’t visible through other testing methods (like traffic interception) become clear once you can see the full code.

Decompiling the Bytecode

The bytecode is usually packaged into specific file types and must be unpacked and then decompiled.

Unpacking the BytecodeJava Applets (.jar files):

Java bytecode is usually packaged in .jar (Java Archive) files.

These files use the zip format, so you can easily unpack them by renaming the .jar file to .zip and then extracting the contents using any zip utility.

Once unpacked, you will find .class files, which contain the Java bytecode.

Silverlight Objects (.xap files):

Silverlight bytecode is stored in .xap files.

These also use the zip format, and the same process can be followed: rename to .zip and extract.

Inside, you’ll find .dll files that contain the bytecode, typically compiled from .NET languages like C#.

Flash Objects (.swf files):

Flash objects come in .swf format and do not need to be unpacked before decompilation.

The .swf file directly contains the necessary bytecode for Flash.

Working on the Client-Side Source Code

- **Input validation or other security-relevant logic**
- **Obfuscation or encryption**
- **"Hidden" functionality, not visible in user interface**
  - **You might be able to unlock it by modifying code**
- **References to server-side functionality you haven't already found via mapping**

Recompiling

Once you've made the desired changes, you need to recompile the modified source code back into bytecode using the relevant development tools for each technology:

•**Java**: Use the javac compiler in the Java Development Kit (JDK) to recompile the .java files into .class files.

•**Flash**: You can use flasm, a disassembler and assembler for Flash bytecode, to modify
the .swf file or Adobe's development tools to recompile the ActionScript code.

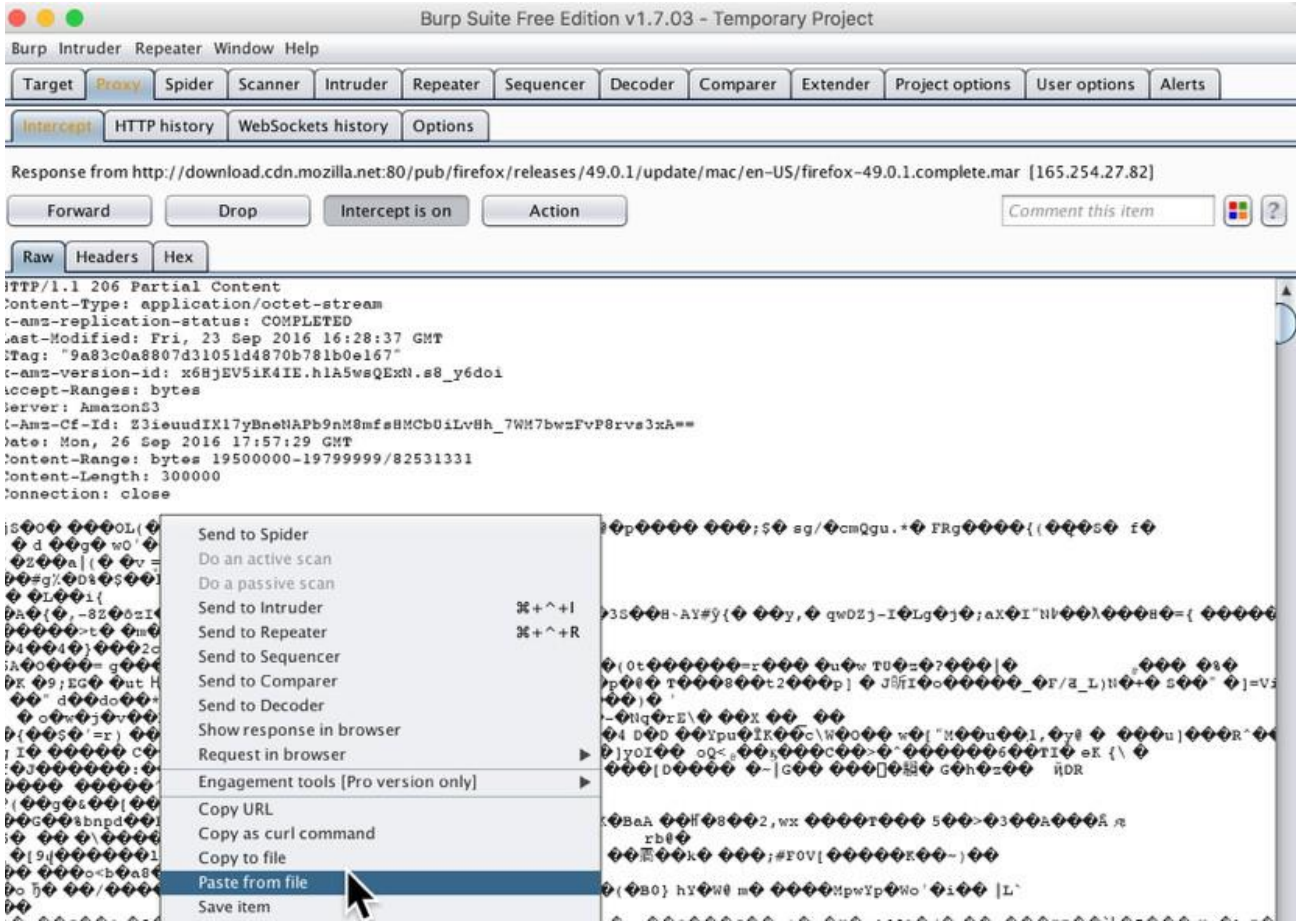
•**Silverlight**: Use **Visual Studio** to compile the modified Silverlight files (.dll).

Loading Modified Component into a Browser

- **Find original component within the browser cache and replace it**
- **Use a proxy to modify the source code of the page that loads the component and specify a different URL pointing to your modified component**
  - **Difficult--may violate same-origin policy**

Loading Modified Component into a Browser

- **Use a proxy to intercept the response containing the executable and replace the body of the message with your modified version**
- **Burp has "Paste from File" option for this**



Response from http://download.cdn.mozilla.net:80/pub/firefox/releases/49.0.1/update/mac/en-US/firefox-49.0.1.complete.mar [165.254.27.82]

```
TTP/1.1 206 Partial Content
ontent-Type: application/octet-stream
-amz-replication-status: COMPLETED
ast-Modified: Fri, 23 Sep 2016 16:28:37 GMT
Tag: "9a83c0a8807d31051d4870b781b0e167"
-amz-version-id: x6HjEV5iK4IE.hlA5wsQExN.s8_y6doi
ccept-Ranges: bytes
erver: AmazonS3
-Amz-Cf-Id: Z3ieuudIX17yBneNAPb9nM8mfsHMCbUiLv8h_7WM7bwzFvP8rvs3xA==
ate: Mon, 26 Sep 2016 17:57:29 GMT
ontent-Range: bytes 19500000-19799999/82531331
ontent-Length: 300000
onnection: close
```

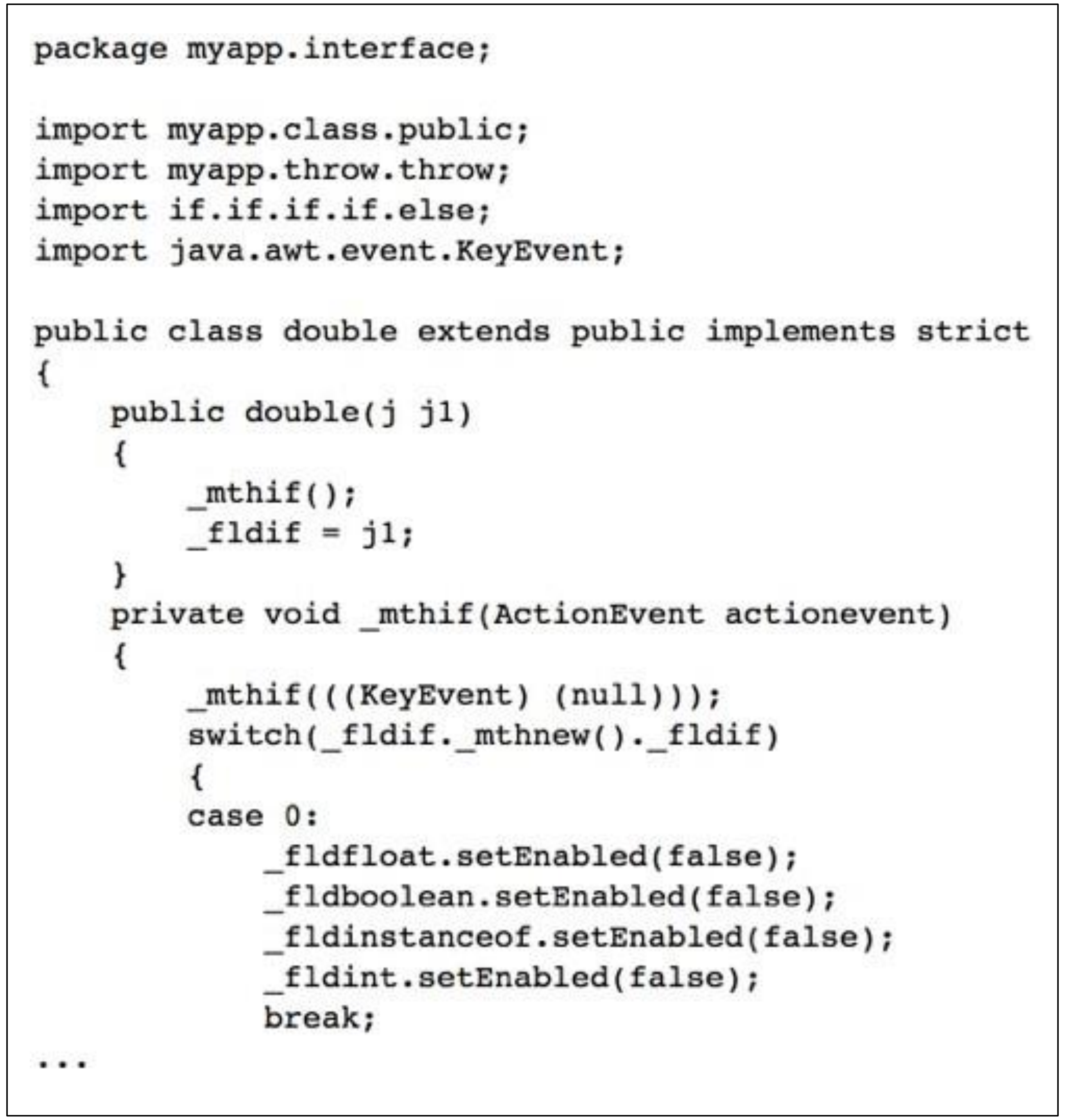
Recompiling and Executing Outside the Browser

Recompiling as a Standalone Program
Change the Execution Context: Normally, the component is designed to
run inside the browser as part of the extension.
To execute it outside the browser, you need to convert it into a standalone program.
The specifics of this depend on the language:

- **Java**: Create a main method to make the applet executable from the command line. This method will serve as the entry point for your program.

Bytecode Obfuscation

- **Difficult to read**



```
package myapp.interface;

import myapp.class.public;
import myapp.throw.throw;
import if.if.if.if.else;
import java.awt.event.KeyEvent;

public class double extends public implements strict
{
    public double(j j1)
    {
        _mthif();
        _fldif = j1;
    }
    private void _mthif(ActionEvent actionevent)
    {
        _mthif(((KeyEvent) (null)));
        switch(_fldif._mthnew()._fldif)
        {
        case 0:
            _fldfloat.setEnabled(false);
            _fldboolean.setEnabled(false);
            _fldinstanceof.setEnabled(false);
            _fldint.setEnabled(false);
            break;
...
```

Obfuscation Methods

•**Meaningless Names**: Class, method, and variable names are replaced with random letters
(e.g., a, b, c), forcing the reverse engineer to study usage patterns to deduce their purpose.
•**Use of Reserved Keywords**: Obfuscators may replace names with language keywords
 (e.g., new, int), making the decompiled code invalid and unreadable.
This prevents recompilation without extensive renaming.

Obfuscation Methods

- **Strip unnecessary debug and meta-information from bytecode, such as**
  - **Source filenames and line numbers**
  - **Local variable names**
  - **Inner class information**
- **Add redundant code that confuses the reverse engineer**

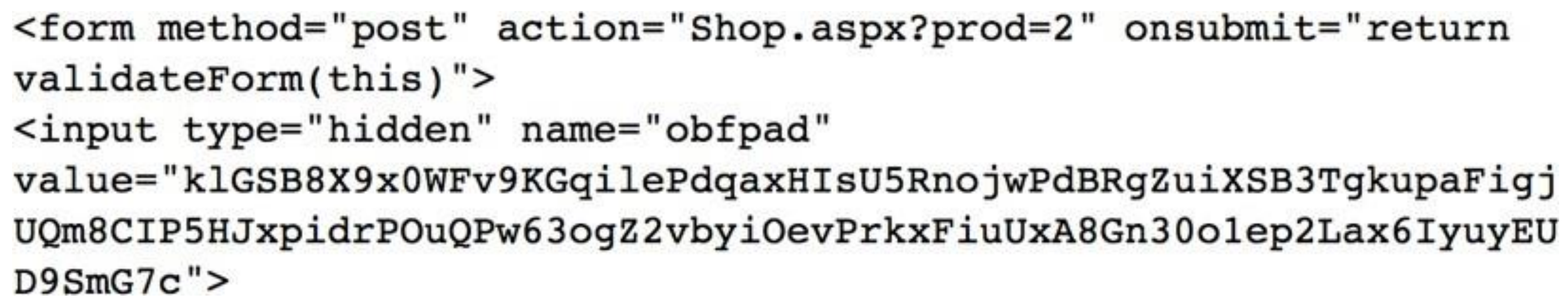
Obfuscation Methods

- **Modify path of execution, make it convoluted**
  - **Using "jump" instructions**
- **Add illegal programming constructs**
  - **Unreachable statements**
  - **Missing "return" statements**
  - **VMs tolerate them, but decompiled source cannot be recompiled without correcting them**

Example: Shopping Java Applet

This portion of the code defines a hidden field named "obfpad"

-



```
<form method="post" action="Shop.aspx?prod=2" onsubmit="return
validateForm(this)">
<input type="hidden" name="obfpad"
value="klGSB8X9x0WFv9KGqilePdqaxHIsU5RnojwPdBRgZuiXSB3TgkupaFigj
UQm8CIP5HJxpidrPOuQPw63ogZ2vbyiOevPrkxFiuUxA8Gn30olep2Lax6IyuyEU
D9SmG7c">
```

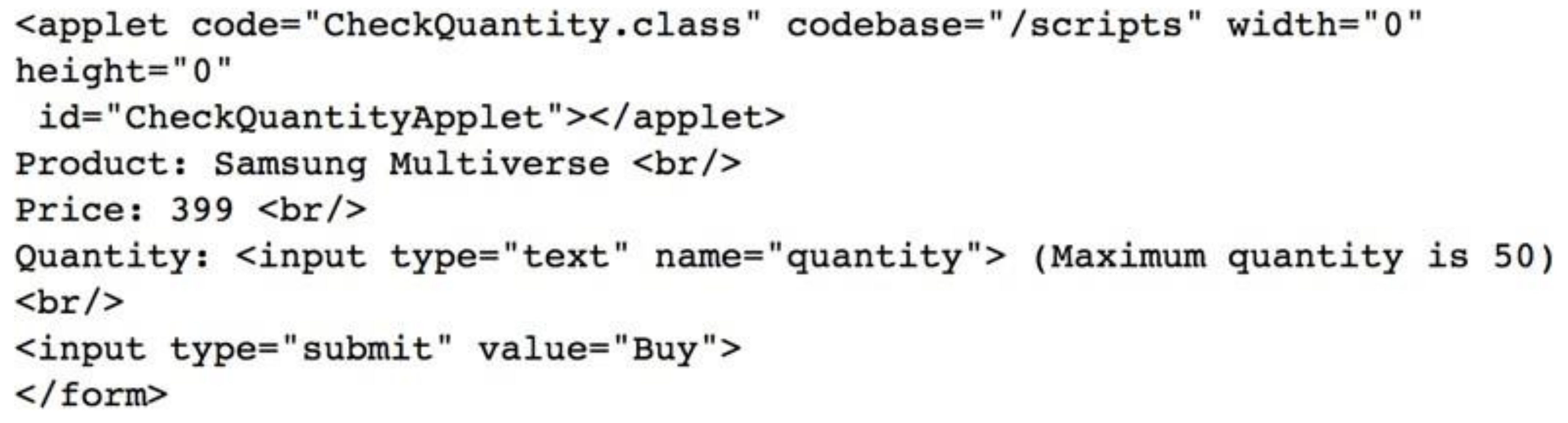
Example Continued

- **JavaScript validator**

```
<script>
function validateForm(theForm)
{
    var obfquantity =
    document.CheckQuantityApplet.doCheck(
    theForm.quantity.value, theForm.obfpad.value);
    if (obfquantity == undefined)
    {
        alert('Please enter a valid quantity.');
        return false;
    }
    theForm.quantity.value = obfquantity;
    return true;
}
</script>
```

Example Continued

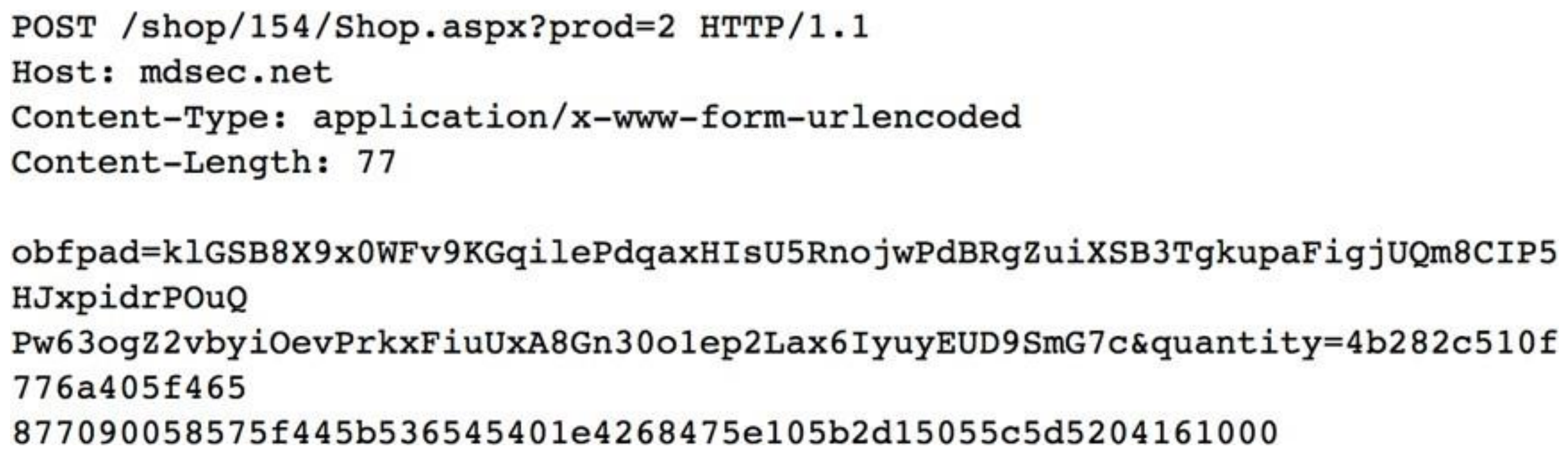
- **This code loads the Java applet "CheckQuantity.class"**



```
<applet code="CheckQuantity.class" codebase="/scripts" width="0"
height="0"
 id="CheckQuantityApplet"></applet>
Product: Samsung Multiverse <br/>
Price: 399 <br/>
Quantity: <input type="text" name="quantity"> (Maximum quantity is 50)
<br/>
<input type="submit" value="Buy">
</form>
```

Example Continued

- **Form is submitted with quantity of 2**
- **It sends this POST request**
- **quantity is obfuscated**

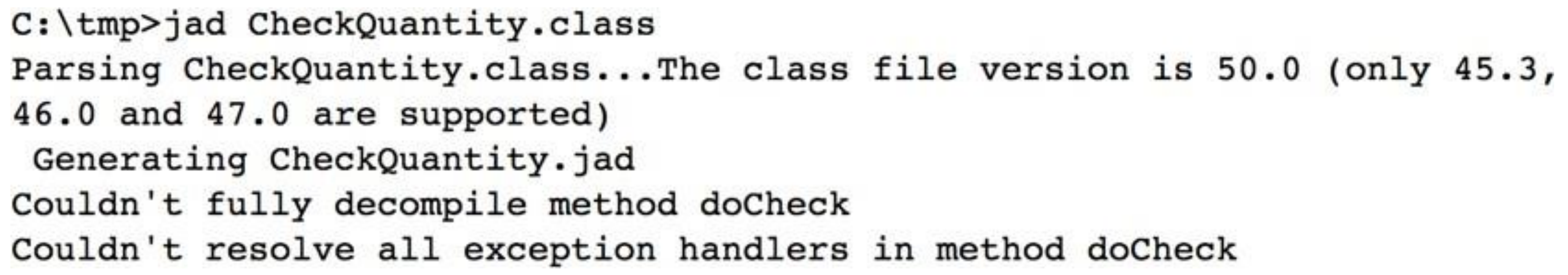


```
POST /shop/154/Shop.aspx?prod=2 HTTP/1.1
Host: mdsec.net
Content-Type: application/x-www-form-urlencoded
Content-Length: 77

obfpad=klGSB8X9x0WFv9KGqilePdqaxHIsU5RnojwPdBRgZuiXSB3TgkupaFigjUQm8CIP5
HJxpidrPOuQ
Pw63ogZ2vbyiOevPrkxFiuUxA8Gn30o1ep2Lax6IyuyEUD9SmG7c&quantity=4b282c510f
776a405f465
877090058575f445b536545401e4268475e105b2d15055c5d5204161000
```

Download Applet and Decompile with Jad

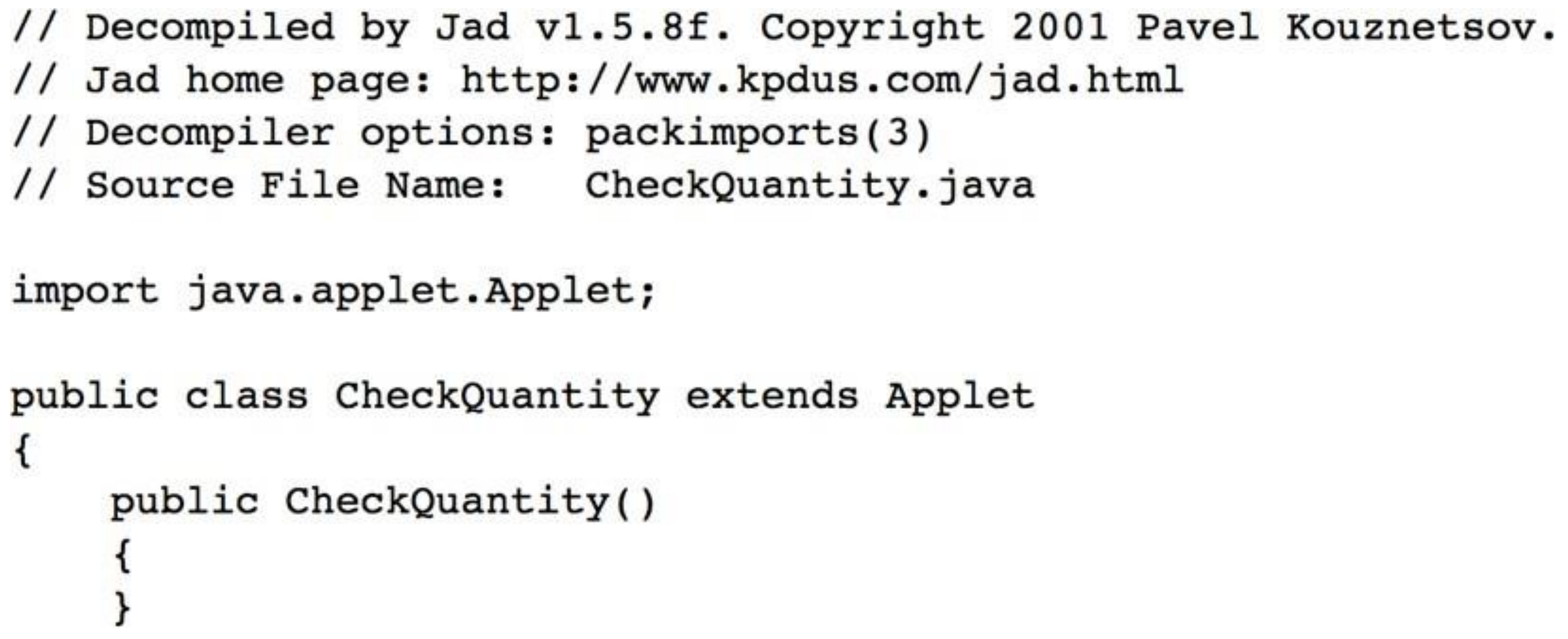
```
/scripts/CheckQuantity.class
```



```
C:\tmp>jad CheckQuantity.class
Parsing CheckQuantity.class...The class file version is 50.0 (only 45.3,
46.0 and 47.0 are supported)
 Generating CheckQuantity.jad
Couldn't fully decompile method doCheck
Couldn't resolve all exception handlers in method doCheck
```

Decompiled Java

- **Header**



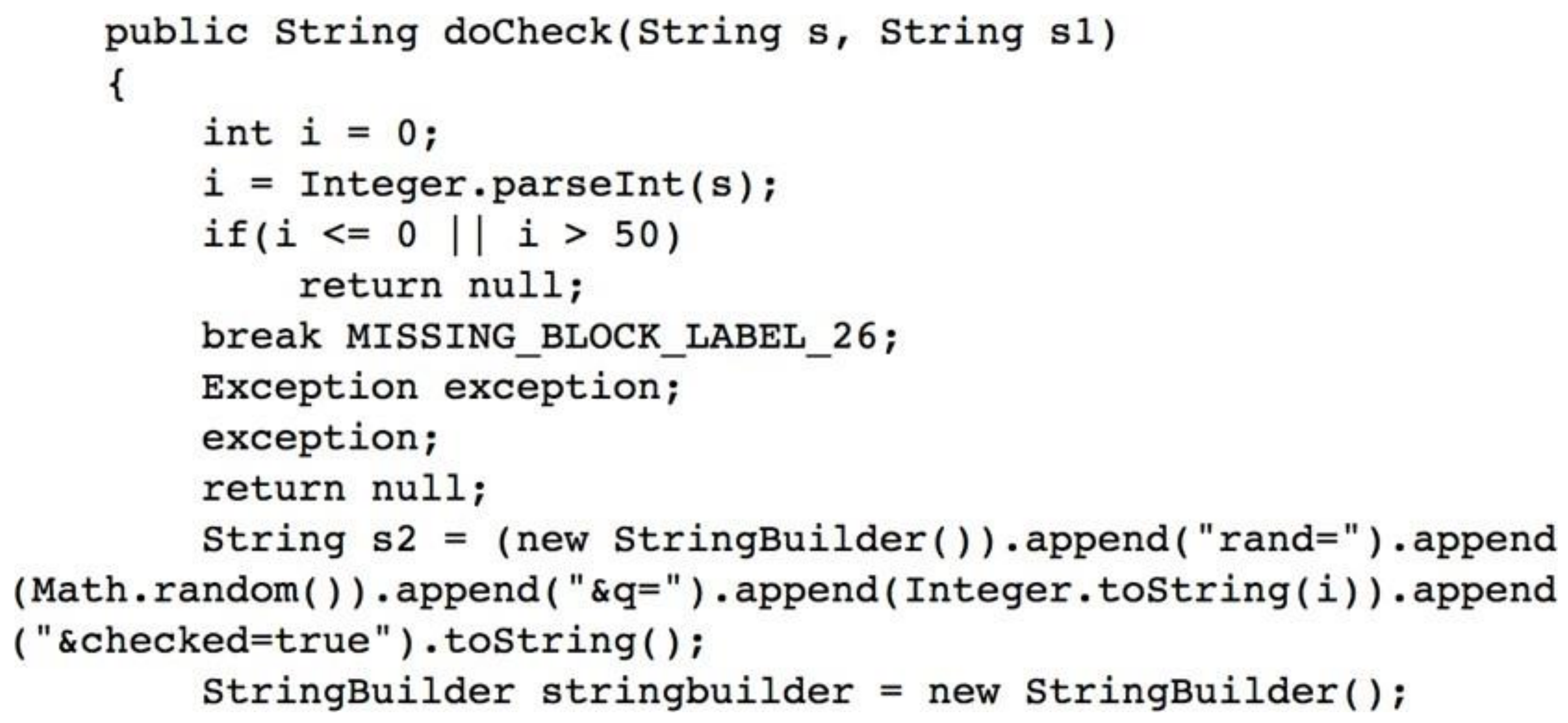
```
// Decompiled by Jad v1.5.8f. Copyright 2001 Pavel Kouznetsov.
// Jad home page: http://www.kpdus.com/jad.html
// Decompiler options: packimports(3)
// Source File Name:   CheckQuantity.java

import java.applet.Applet;

public class CheckQuantity extends Applet
{
    public CheckQuantity()
    {
    }
```

Decompiled Java

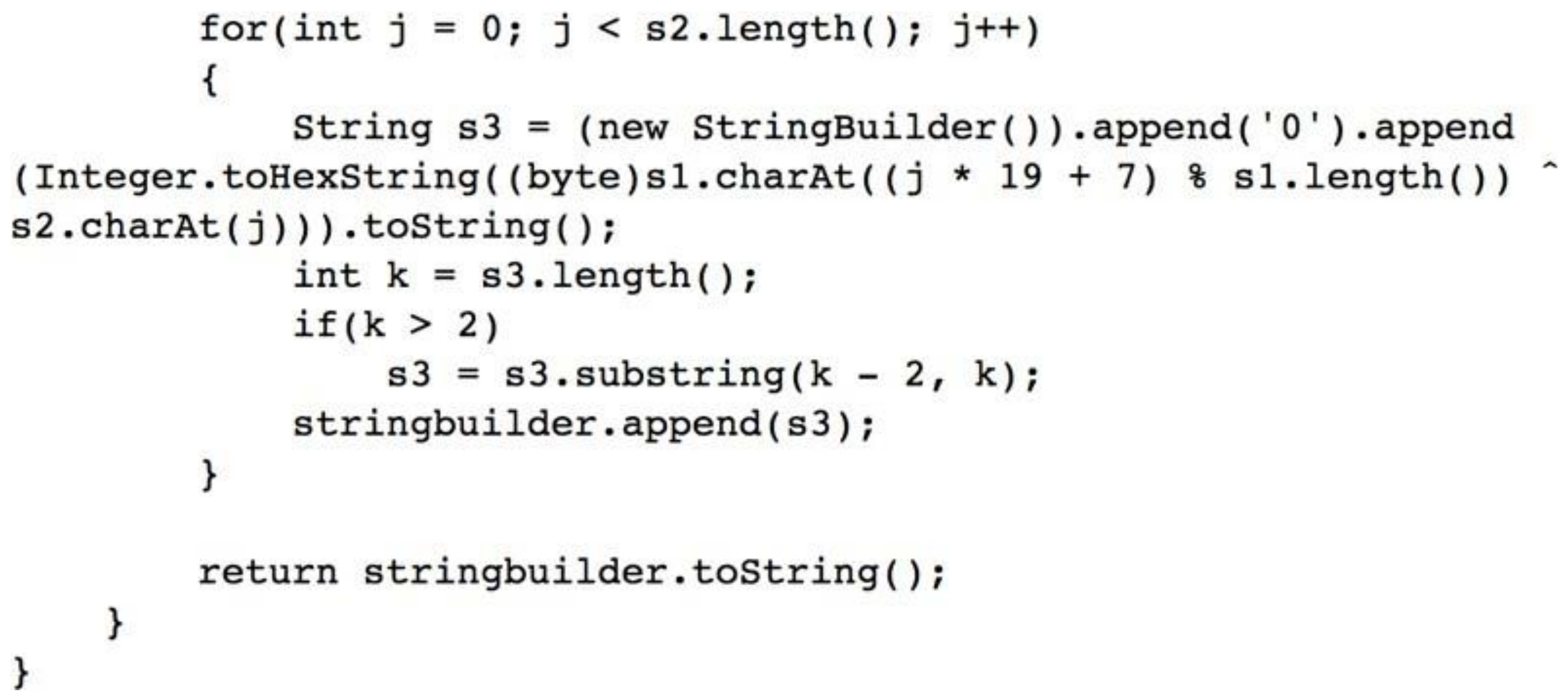
- **Verifies tha qty is between 0 and 50**
- **Prepares a string to hold obfuscated text**



```
    public String doCheck(String s, String s1)
    {
        int i = 0;
        i = Integer.parseInt(s);
        if(i <= 0 || i > 50)
            return null;
        break MISSING_BLOCK_LABEL_26;
        Exception exception;
        exception;
        return null;
        String s2 = (new StringBuilder()).append("rand=").append
(Math.random()).append("&q=").append(Integer.toString(i)).append
("&checked=true").toString();
        StringBuilder stringbuilder = new StringBuilder();
```

Decompiled Java

- **XORs text with obfpad**



```
        for(int j = 0; j < s2.length(); j++)
        {
            String s3 = (new StringBuilder()).append('0').append
(Integer.toHexString((byte)s1.charAt((j * 19 + 7) % s1.length()) ^
s2.charAt(j))).toString();
            int k = s3.length();
            if(k > 2)
                s3 = s3.substring(k - 2, k);
            stringbuilder.append(s3);
        }

        return stringbuilder.toString();
    }
}
```

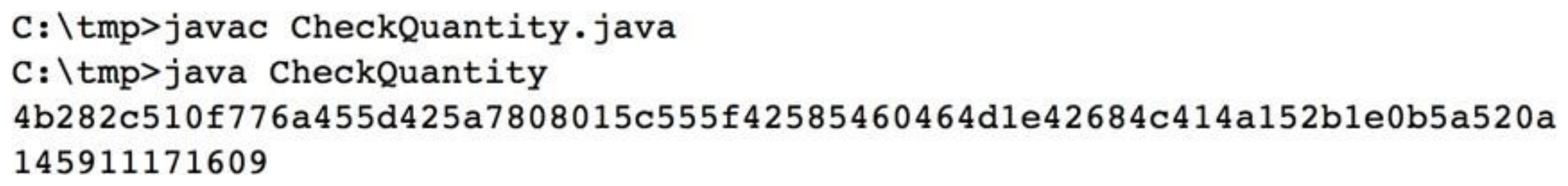
Modified Applet

- Remove input validation from "doCheck"
- Add a "main" method that calls "doCheck" and prints the obfuscated result
- Code on next page outputs obfuscated string for quantity of 999
- Try various values, including strings, to find vulnerabilities

```java
public class CheckQuantity
{
    public static void main(String[] a)
    {
        System.out.println(doCheck("999",
"klGSB8X9x0WFv9KGqilePdqaxHIsU5RnojwPdBRgZuiXSB3TgkupaFigjUQm8CIP5HJxpi
drPOuQPw63ogZ2vbyiOevPrkxFiuUxA8Gn30olep2Lax6IyuyEUD9 SmG7c"));
    }

    public static String doCheck(String s, String s1)
    {
        String s2 = (new StringBuilder()).append("rand=").append
(Math.random()).append("&q=").append(s).append
("&checked=true").toString();
        StringBuilder stringbuilder = new StringBuilder();
        for(int j = 0; j < s2.length(); j++)
        {
            String s3 = (new StringBuilder()).append('0').append
(Integer.toHexString((byte)s1.charAt((j * 19 + 7) % s1.length()) ^
s2.charAt(j))).toString();
            int k = s3.length();
            if(k > 2)
                s3 = s3.substring(k - 2, k);
            stringbuilder.append(s3);
        }
    return stringbuilder.toString();
    }
}
```

Recompile with javac



```
C:\tmp>javac CheckQuantity.java
C:\tmp>java CheckQuantity
4b282c510f776a455d425a7808015c555f42585460464d1e42684c414a152b1e0b5a520a
145911171609
```

Compile the modified applet and run it to generate an obfuscated string for any quantity.
Intercept the HTTP request using a proxy tool (like Burp Suite), and replace the obfuscated quantity with the one generated by your modified applet

Native Client Components

- Some actions cannot be done from a sandbox
    - Verifying that the user has up-to-date antivirus
    - Verifying proxy settings
    - Integrating with a smartcard reader
- For these, native code components like ActiveX are used; they run outside the sandbox

Reverse-Engineering Native Code

- **OllyDbg to debug**
- **IDA Pro or Hopper to disassemble**

Handing Client-Side Data Securely

- **Don't send critical data like prices from the client**
  - **Send product ID and look up price on server**
- **If you must send important data, sign and/or encrypt it to avoid user tampering**
  - **May be vulnerable to replay or cryptographic attacks**

Validating Client-Generated Data

- **All client-side validation methods are vulnerable**
  - **They may be useful for performance, but they can never be trusted**
- **The only secure way to validate data is on the server**

Logs and Alerts

- **Server-side intrusion detection defenses should be aware of client-side validation**
- **It should detect invalid data as probably malicious, triggering alerts and log entries**
- **May terminate user's session, or suspend user's account**