**Secure Coding Practice Workshop**

Presented by Security Ninjas

(Bhakti Bohara, Deepanjana Gupta, Deepika Peringanji, Shivani Sharma)

**Agenda**

This curriculum focuses on introducing developers to secure coding practices so that their code is nearly bug-free at least on the security side! It will include security concepts that a developer should know when (s)he starts developing an application from database up till the front end.

Some of the concepts discussed would be

* Database security
* Memory and file management
* System configuration
* Input validation
* Output encoding
* Error handling
* Web session management

**Key Takeaways**

* Coding anti-patterns in backend and frontend code leading to these vulnerabilities
* Security vulnerabilities arising from these bad practices
* Solutions and tips on how to avoid these practices.

**Logistics**

The class is divided into different stages, each stage deals with a commonly observed vulnerability in the code we write. After introducing the student to the vulnerability, he/she will be asked to attack the code snippet, once the vulnerability will be spotted and attacked, the student will have to modify the snippet to make it secure.

**Prerequisite**

* Mac / Linux - or virtual box for terminal access
* As different parts of a single application sometimes require coding in multiple languages, this workshops requires the student to have a basic working knowledge of SQL, C, C++ , Javascript or any other scripting language.
* Sqlite installed on the system. Installation files can be provided with the workshop material

**Session - 1 : Database security**

* **Access Control:** Access control measures should be in place. Secure credentials which should be never hardcoded in the application should be used. Application should use lowest possible levels of the privilege.

Stored procedures are ideal to abstract data access

* **Data Leaks:**  Though mostly part of the backend, databases do contain networking interfaces. Hacker can capture and exploit traffic on these interfaces. SSL or TLS encrypted communication should be in place.
* ***SQL Injection:*** This attack consists of maliciously running a sql query on the database by putting it in one of the user inputs of the intended query. The malicious query can do damages like retrieving unauthorised data, changing permissions on the database, deleting tables, changing schema etc.

eg:

*uname = <user-input>*

*pwd = <user-input>*

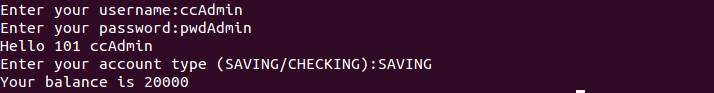
*SELECT \* FROM users WHERE username=<uname> AND password= <pwd>*

*SELECT \* FROM users WHERE username = 'jeff' AND password = 'afhasfha'*

What if pwd is “'afhasfha' AND 1” !!

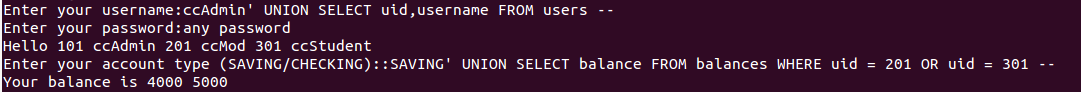
**Exercise - 1 : sqlInjection**

**A normal run:**



***<For Instructor>*** Tell students about the table schema (refer to createTable.py) and ask them to retrieve balance of another user using sql injection

**An injected run:**



**Session- 2 : Memory and File Management**

* **Memory Management**

Depending on the language we’re using to code, applications may be vulnerable to attacks on the memory. C has a myriad of functions in several library packages that do not validate the number of bytes it is going to work with, for example, *strcpy, strcat, gets, scanf, sprintf*

Attackers can enter strings larger than the allocated memory to hold them and hence corrupt surrounding memory leading to unpredictable program behavior.

* **Stack Smashing**

When a function is called in C, the arguments and local variables and callee variables are saved on the stack in case of a 32-bit architecture (different for a 64-bit architecture, refer to Appendix A). By overwriting the return address of the caller an attacker can corrupt the stack, This is called as *stack smashing*. It disables the program to ever return.

Best practice is to use functions that are not vulnerable to stack smashing as they read number of bytes to be worked with eg: *strncpy, strncat, fgets, snprintf etc…*

* **Integer Overflow**

This occurs as a result of an operation involving two or more integers that result in an integer whose size exceeds the variable that it is supposed to be stored in.

Eg: if a 32 bit signed integer variable is being used, the maximum value it can store is 7FFFFFFF and a user might think if he adds 1 it will result in positive 80000000 but since it is a signed integer it will actually result in the most negative integer and this may not be the value an unmindful user is expecting.

* **Format String Exploit**

This attack occurs when the submitted data of an input string is evaluated as a command by the application. In this way, the attacker could execute code, read the stack, or cause a segmentation fault in the running application, causing new behaviors that could compromise the security or the stability of the system. Functions like *sprintf* are vulnerable to these attacks and their safer alternatives like *snprintf* should be used.

* **File Management**

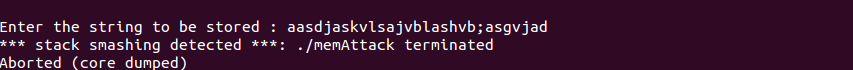
Take care of file permissions while creating them thus not allowing unauthorised downloads. Unauthorised uploads should also be checked and uploaded files should be scanned for viruses. Also, the access control of the file directory should be strictly maintained.

**Exercise - 2: memoryAttacks**

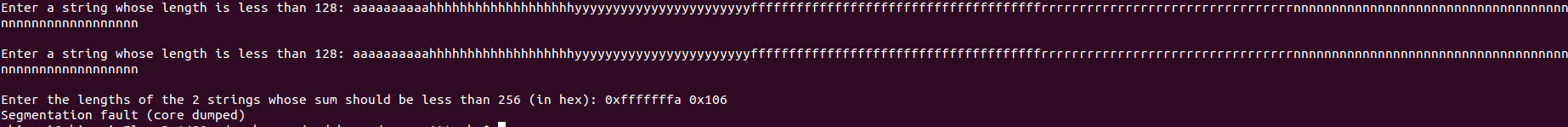
Open overflow.c and analyse the code. Find the vulnerabilities in the code and take advantage of these to smash the stack.

Hint : There are 2 ways you can do this. (Direct buffer overflow or by integer overflow)

**Buffer overflow:**



**Integer overflow:**



Solution : If for the string to be entered anything more than 24 bytes (for 64-bit architecture: refer to how stack is used in different architectures) are entered, buffer overflow occurs.

Also if strings of lengths more than 128 is entered unlike what is asked for, integer overflow can be done and the validity condition can be made to fail causing buffer overflow.

**Session - 3 : Cryptographic Practices**

Using cryptography to encrypt/decrypt data doesn't make it all secure! There are many things that are to be taken care of while using crypto libraries. It is important to understand how these libraries work.

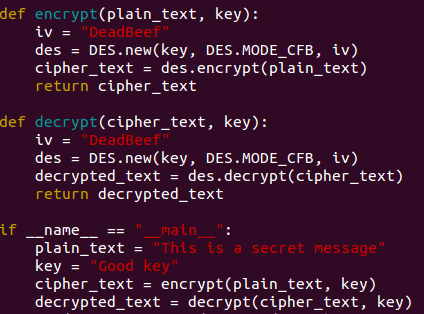
Any encryption/decryption algorithm needs good keys to work efficiently. Small keys make it easy to break the encryption. Choosing a good key isn’t it all. Same content encrypted with good keys will still lead to same encrypted content. To avoid this, some algorithms like AES choose an initialization vector (IV). It is again important to generate this IV in a random fashion.

**Exercise - 3 : Cryptography Constant IV**

The code snippet in crypt\_1.py tries to encrypt and decrypt a given message and tests if its successful at that.

Try running crypt\_1.py - python crypt\_1.py

It works but it’s not a safe coding practice. Now identify the bad practice and rectify it.



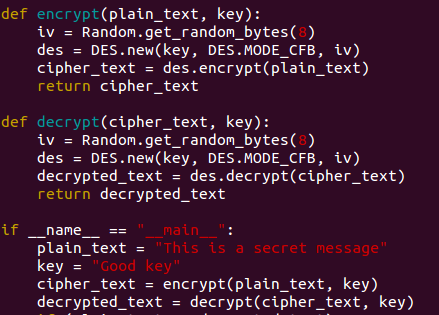
crypt\_1.py uses the same hardcoded IV for any encrypt/decrypt instance which isn't secure. If the attacker finds out the IV once, he can use it for all subsequent attacks.

**Exercise - 4 : Cryptography Functions’ Order**

Try python crypt\_2.py. This script rectifies the problem of hardcoded IV by using crypto’s “Random” library.

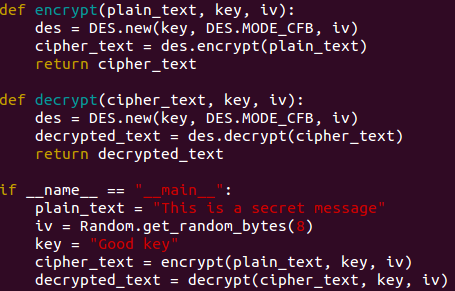
But crypt\_2.py doesn't work at all!

**Buggy code:**



The error lies in randomly generating IVs in encrypt and decrypt function. We should use the same IV for both encrypt and decrypt.

**Correct code:**



**Session - 4 : Session Management**

HTTP is a stateless protocol. Every communication between client and server is independent. As a result it gets difficult to get a continuity in the information exchange. A session provides the ability to set variables such as local information and access permissions which will be further applied to every web interaction.

*What is a cookie?*

A web cookie is data stored in the web browser by the website. It can be set by the client or the server.

Security of the cookie will depend on the security of the website that has set it and the browser. Vulnerabilities may give way for hackers to read cookie data and further gain access to the website to whom the cookie belongs.

*Eg : John:Smith ;* ***2016-08-16T16:44:41+00:00***

*username = John*

*Password = codechix1234*

*Frames : yes*

*Why are cookies used?*

* **Personalization** : Search Engine, Social networks use cookies to record user search patterns and present advertisements that reflect your search.
* **Security** : Most websites store cookies in user’s browser than storing it in web servers. If web servers are hacked , user’s informations is stolen. The optimal method on where the user’s information has to be stored is debatable.
* **Offload** : A cookie will take up less space on user’s browser, but could take significantly large space when sent back to server. In order to render efficient personalisation, it is important that the cookies are accessed without delay.
* **Privacy** : Websites provide cookies after user’s have identified themselves. By using cookies, websites can identify users without having disclosure of identity.

Points to remember..

* Do not store sensitive information in the cookie.
* Restrict access from other applications by adding domain names
* To make it inaccessible to javascript, make it HTTPOnly
* Use SSL to transfer the cookie in secure manner.

**Exercise - 5 : sessionMgmt**

A simple script is being provided to save a cookie on the client machine. There is no connection to a server.



Tasks

*The script can be copied on emulator on w3 schools to code and run parallely.*

1. Restrict the cookie saved by adding following restrictions:
   1. Path
   2. Domain
   3. Max-Age
2. Use URL Encoding to eliminate whitespaces - encodeURIComponent()
3. Use secure attribute to change protocol to https
4. Use HTTPOnly attribute to instruct browsers to not allow javascript to access cookies via document.cookie.object module . This also prevents XSS access.
5. Write a simple server to send session id in a cookie through headers.
6. On logout, invalidate the session id.

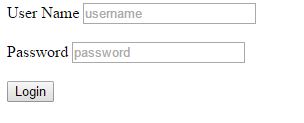
**Session - 5 : Input Validation**

At the front end, input validation is extremely important as it can help prevent XSS and CSRF attacks (to learn more about these refer to Appendix B)

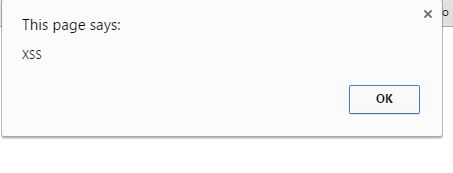
While writing an interface for the user, it is important to sanitize the input entered by the user. This can be done by filtering and validating the input by two ways - blacklisting (disallowing certain expressions to be part of the input) or whitelisting (allowing only specified expressions in input fields). Input can also be encoded.

**Exercise - 6 inputValidation**

You have been provided a form with 2 input fields. You can open it (input\_valid.html) in your browser.



Your job is to display the string “XSS” by entering a crafted query in the input field.



**Session - 6 : Output Encoding**

Output encoding is required so that any potentially dangerous characters/content in the output is safe and sanitized. Source of the output can be servers, users, clients, networks and databases. Output encoding is primarily associated with preventing Cross-site Scripting (XSS) attacks.

Commonly used encodings usually leave safe characters like alphanumerics as it is while sensitive characters like markup characters (<, >, ”, ’) are encoded.

|  |  |
| --- | --- |
| **Html Code** | **Encoded output** |
| <html>  <head>  <script type="text/javascript">  function validate() {  var uname = document.forms["logincheck"]["user"].value;  alert("Welcome "+ uname);    }  </script>  </head>  <body>  <form name="logincheck">  <label>User Name</label>  <input type="text" name="user" placeholder="username">  <br> </br>  <label>Password</label>  <input type="password" name="password" placeholder="password">  <br> </br>  <input type="submit" value="Login" onclick="validate();" />  </form> | &lt;html&gt;  &lt;head&gt;  &lt;script type=&quot;text/javascript&quot;&gt;  function validate() {  var uname = document.forms[&quot;logincheck&quot;][&quot;user&quot;].value;  alert(&quot;Welcome &quot;+ uname);    }  &lt;/script&gt;  &lt;/head&gt;  &lt;body&gt;  &lt;form name=&quot;logincheck&quot;&gt;  &lt;label&gt;User Name&lt;/label&gt;  &lt;input type=&quot;text&quot; name=&quot;user&quot; placeholder=&quot;username&quot;&gt;  &lt;br&gt; &lt;/br&gt;  &lt;label&gt;Password&lt;/label&gt;  &lt;input type=&quot;password&quot; name=&quot;password&quot; placeholder=&quot;password&quot;&gt;  &lt;br&gt; &lt;/br&gt;  &lt;input type=&quot;submit&quot; value=&quot;Login&quot; onclick=&quot;validate();&quot; /&gt;  &lt;/form&gt;  &lt;/body&gt;  &lt;/html&gt; |

**Session - 7 : Error Handling**

**Prerequisites:** Install Valgrind on your machine

When the user enters an invalid input, the error should be handled gracefully and should not reveal any sensitive information about the underlying application, system, server etc.,

Example: During authentication, in case of wrong password, it is beneficial to throw error as “Wrong username/password” instead of “Wrong password”.

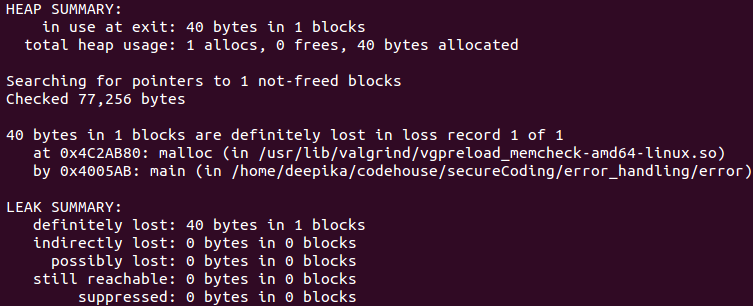
Do not give information to user that is not needed to him!

In case of failure, any resources allocated should be released before exiting the program else the system will go into an inconsistent state.

**Exercise - 7 Free Memory**

Observe the code in error.c and identify the bad practices. You can install Valgrind to check for the errors

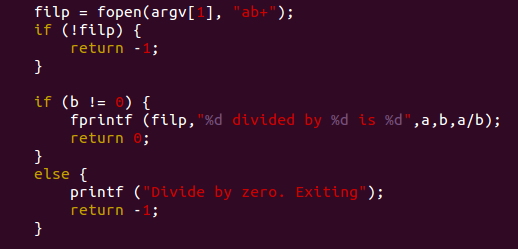
valgrind --leak-check=full -v ./memory\_handling



**Exercise - 8 Cleanup**

This exercise helps in identifying the resources allocated and releasing them.

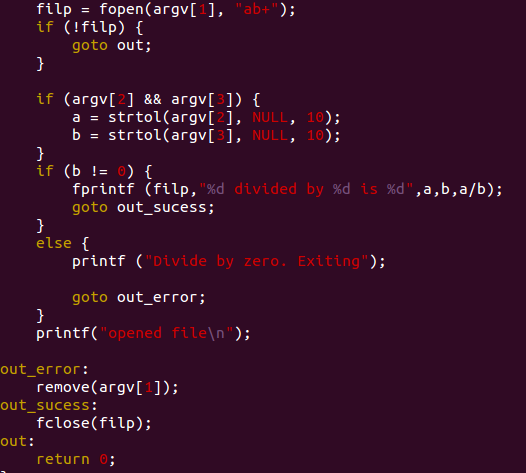
Buggy Code:



In case of error, that is division-by-zero the program exits without closing/deleting the file

These kind of practices leave traces/temporary files in the system which the attacker can access and learn about things he is not supposed to!

Solution code:



**References**

1. <https://www.owasp.org/images/0/08/OWASP_SCP_Quick_Reference_Guide_v2.pdf>
2. <http://www.w3schools.com/>
3. http://www.opinionatedgeek.com/Errors/Unknown.aspx?aspxerrorpath=/dotnet/tools/htmlencode/encode.aspx

**Appendix A**

For 32 bit architecture, refer to <http://www.tenouk.com/Bufferoverflowc/Bufferoverflow2a.html>

For 64 bit architecture, refer to <http://eli.thegreenplace.net/2011/09/06/stack-frame-layout-on-x86-64/>

**Appendix B**

For XSS details, refer to <https://www.owasp.org/index.php/Cross-site_Scripting_(XSS)>

For CSRF details, refer to <https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)>