Challenge SSTIC 2026

Solution de travail

Assistance à l'Agence Bretonne de la Sécurité des Systèmes d'Information

Axël Bessade

Enedis DSI

12 mai 2026

Table des matières

Introduction

Résumé

Ce document présente la résolution complète du challenge SSTIC 2026.

Message initial

```
Hello analyst,
Following an alert from the ABSSI (Agence Bretonne de la Sécurité des Systèmes d'Information), we are suspecting
a compromise of one of our contractors. An extract, containing a dubious network traffic, has been supplied.
Could you please have a look at it?
As usual, we are looking for an analysis of the exchanges, and any IOCs if relevant.
Please be careful with contained data, if any. The, maybe, compromised contractor is working on a highly
sensitive infrastructure, all information related to this system MUST BE reported at the earliest opportunity.
Thanks for your assistance and your discretion,
Incident Dispatcher - Investigation des Moyens et Plateformes Sous-traitées
```

Le point de départ est un extrait de trafic réseau suspect.

Le challenge SSTIC 2026 débute ainsi par une demande d'assistance émise par l'ABSSI, l'Agence Bretonne de la Sécurité des Systèmes d'Information.
La page officielle du challenge est disponible à l'adresse suivante : Challenge 2026
Le premier artefact fourni est une capture réseau : client_capture.pcapng

L'objectif annoncé est double : analyser les échanges présents dans cette capture et identifier, si possible, les éléments liés à la compromission du prestataire. La suite montre progressivement que ce prestataire a accès à des informations relatives à SAFE, un système d'arme protégé par une architecture à diode.

Cette solution suit les intitulés publics des épreuves :

— Step 0 : *linenoise*

— Step 1 : *vibe malwaring*

— Step 2 : *a core lock*

— Step 3.1 : *lobster128 parameters*

— Step 3 : *overflowing faults*

— Step 4 : *dancing in shadow*

— Step 5 bonus : *dumping through my screen*

Chaque section présente le contexte de l'étape, les observations déterminantes, la méthode employée, puis le résultat obtenu. Les extraits de code et de sortie sont volontairement limités aux éléments nécessaires pour reproduire le raisonnement.

Le format retenu reste compact, avec une page par étape. Après un mois de travail sur le challenge, c'est un compromis raisonnable entre lisibilité, temps disponible et énergie restante.
Merci à Pierre Bienaimé pour l'inspiration de ce format rapide, simple et efficace. Je n'ai pas trouvé de contact pour le remercier directement, donc je le fais ici.

Conformément aux règles du challenge, je précise également avoir utilisé un LLM pour m'aider à rédiger, relire, reformuler, améliorer et corriger ce document. Et également pour éviter de réveiller Bernard Pivot avec mes fautes de français.

1 | Step 0 : linenoise

Le challenge commence avec `client_capture.pcapng`, fourni directement depuis la page SSTIC.
La capture contient uniquement de l'UDP entre `203.0.17.102` et `203.0.2.95:443`. Sur le papier, cela ressemble donc à du QUIC valide.

Un premier passage rapide sur les paquets montre dix connexions QUIC v1 parallèles. Chaque connexion correspond à un ordre serveur différent : `init_crypto`, puis plusieurs `get_module`.
Comme le payload QUIC est chiffré, les champs encore visibles deviennent naturellement des candidats intéressants pour un canal caché. Le Destination Connection ID est particulièrement adapté : il est présent dans chaque paquet, varie normalement entre connexions, et son contenu n'est généralement pas inspecté fonctionnellement par les outils réseau classiques.

La stéganographie est bien là. L'approche reste discrète : Wireshark décode le trafic comme du QUIC valide, sans signaler d'anomalie particulière. Le canal caché est entièrement porté par des métadonnées protocolaires légitimes.
Les commandes serveur sont encodées dans les DCID des paquets serveur vers client, et les réponses du client dans les DCID des paquets client vers serveur.
Les retransmissions QUIC rajoutent un peu de bruit, donc il faut dédupliquer les DCID consécutifs identiques avant de concaténer les octets.

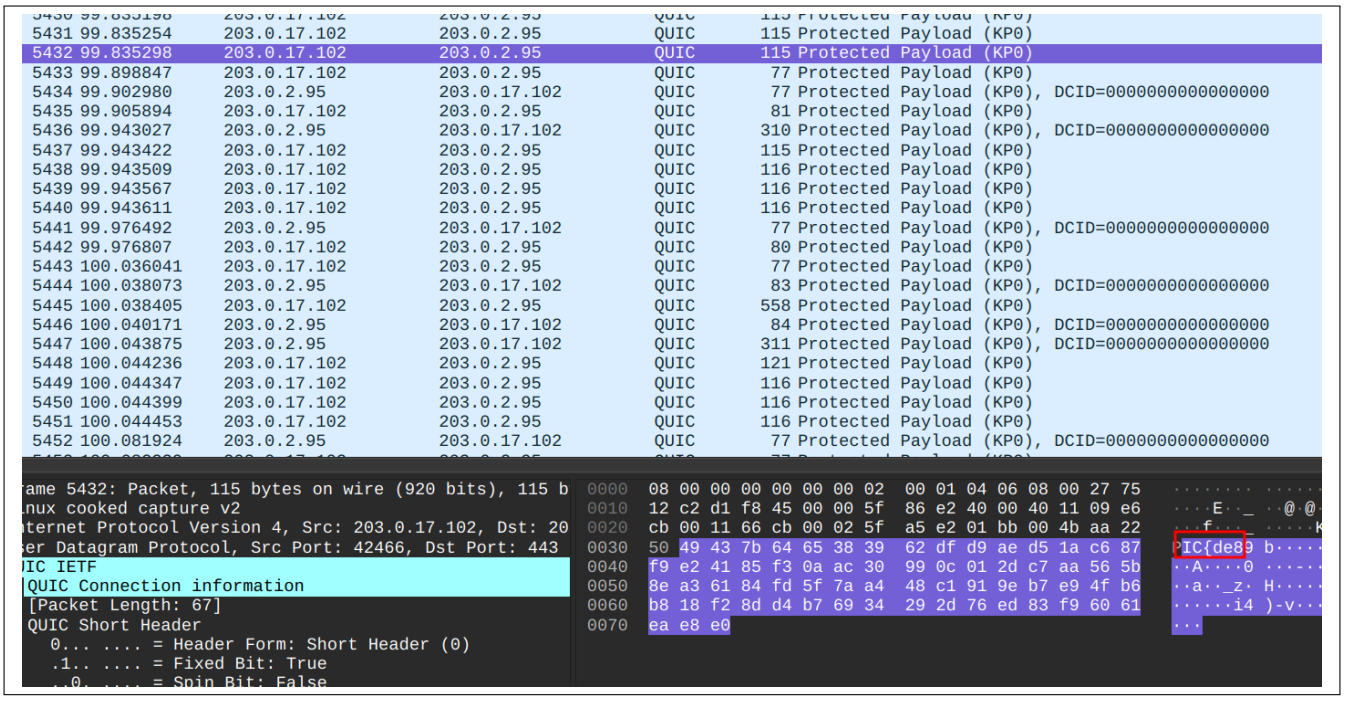


FIGURE 1.1 – Visualisation Wireshark des paquets QUIC et du champ Destination Connection ID utilisé comme canal caché.

L'extraction des flux reconstruits permet de récupérer plusieurs modules Python du client : `dga`, `filer`, `quic`, `comm`, `order`, `client`, ainsi qu'une configuration chiffrée `config.enc`.
Le module `utils.py` contient directement le premier flag. Le déchiffrement des paquets QUIC Initial a également permis de valider l'analyse du protocole, mais l'information utile pour l'étape se trouvait dans le canal caché porté par les Connection IDs.

Flag Step 0

`SSTIC{de89bf301aa2ef9f9a61486d26c7b81424bcf5b838f98dde}`

2 | Step 1 : vibe malwaring

Une fois les modules récupérés, le plus simple est de reconstituer le client localement.
La configuration n'est pas en clair : `config.enc` est chiffré en AES-CBC avec une clé issue de `Random(int(time.time())).randbytes(32)`. Comme on dispose de la fenêtre temporelle de la capture, une recherche exhaustive sur les timestamps candidats est adaptée.

```
capture start : 1771542009
capture end   : 1771542182
seed trouve   : 1771542017
```

La configuration déchiffrée donne le C2, le filer HTTP, et surtout la seed DGA :

```
c2_base_domain = "203.0.2.95"
c2_base_port = 443
filer_base_ip = "51.15.164.185"
filer_base_port = 80
filer_dga_seed = "9a04ca81d4a8bb16ee782e90984c7f4d55cb21bafa3e35e720628a400aae6e91"
```

Le DGA est ensuite très classique : on mélange la seed avec l'année, le mois et la semaine ISO, puis on utilise le hash comme graine de PRNG pour générer un nom de 16 caractères.
Pour la semaine du challenge, cela donne le filer :

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

Le service de stockage associé à cette étape apporte les principaux éléments de contexte pour la suite. On y retrouve des documents internes, des mails, une archive crypto, les binaires SAFE, des notes sur ShadowStack, une base utilisateur, une diode source/destination et les premières traces de SiviHaKerez.
Le scénario commence à se dessiner : un groupe hacktiviste a compromis un prestataire, récupéré des documents Aegis, puis s'en est servi pour attaquer SAFE.

Le fichier `flag.txt` du filer donne le flag de l'étape.
Le `readme.txt` contient aussi l'URL de l'étape suivante.

Ces éléments déplacent l'analyse vers le mécanisme de transfert contrôlé par la diode et vers le format d'archive accepté par celle-ci.

Flag Step 1

`SSTIC{c8abe2747c3f4a75d4d01ed5e3f9f3ebceae4cb4995ebddccdf41cdf7a42807d}`

3 | Step 2 : a core lock

L'URL donnée par le filer spawn une instance de SAFE :
http://51.15.164.185/step/5bc47fb5b3fb831ee96884387fd16871
Le système est découpé en deux morceaux. Côté exposé, `diode_src` reçoit des archives `.sa` via SFTP. Côté scellé, `diode_dst` reçoit les messages validés et exécute les handlers applicatifs.
Dans la documentation d'exploitation, on trouve les vrais identifiants SFTP :

```
login    : diode_client
password : {Thisp@ssw0rdShouldN0tB3GUESSED}
```

Le SFTP expose trois répertoires utiles : `archive/`, `in/` et `log/`. Les archives historiques ne sont pas lisibles directement, mais leurs logs le sont.
Ils racontent une chronologie assez claire : les mises à jour légitimes passent, puis SiviHaKerez pousse `pown_key.sa`, puis `hell_fire.sa`. Le remplacement de clé est donc probablement la bascule.

Il faut ensuite comprendre le format `.sa`. Une archive contient un header, un CRC64, des tags, un paquet compressé LZO, une signature Lobster256 et un secret.
`diode_src` vérifie tout ça dans l'ordre : CRC, header, tags, paquet, signature, secret, puis transmet vers la destination.

```
0x00 magic
0x08 crc64(data[16:])
0x10 pkg_offset
0x18 pkg_decompressed_size
0x20 sig_offset
0x28 secret_offset
0x30 tags, puis pkg, sig, secret
```

Le bug qui donne l'étape n'est pas dans la crypto. Il est dans une fonction de debug.
Quand le tag `_SHA256` est présent, le binaire lance un `sha256sum data/in/<filename>` avec `popen()`, sans sanitizer le nom du fichier. Le SFTP interdit les slashs, mais pas besoin de slash pour faire du shell.

```
filename = ";cd data&&cat flag.txt;#.sa"
```

Le `#` commente la fin de la commande, et le `&&` remplace gentiment les slashs interdits.
L'injection permet de lire `flag.txt`, puis de récupérer les archives historiques. On confirme alors que `prod_maj_key.sa` contenait la clé originale avec la clé privée, et que `pown_key.sa` a installé la clé publique SiviHaKerez.

Flag Step 2

`SSTIC{fa0405ed24364461327146760b57051767a19a36d944335ae4449615ca60ddd7}`

4 | Step 3.1 : lobster128 parameters

Entre le bug de `diode_src` et la forge de signature finale, une étape intermédiaire est consacrée à Lobster. L'épreuve *Step 3.1 : lobster128 parameters* demande de ne plus traiter l'algorithme comme une boîte noire : il faut retrouver les paramètres de la courbe elliptique et comprendre le schéma de signature utilisé par SAFE.

La signature est une variante EC-KCDSA. La convention est un peu piégeuse : la clé publique est de la forme `Q = d^-1 * G`, pas `Q = d * G`.
Les clés publiques sont stockées sur 36 octets :

```
00 02 2d || X sur 32 octets || parite
```

Pour résoudre les paramètres de courbe, j'ai utilisé les points précompilés du binaire Lobster256. Avec les relations de doublement et d'addition, on peut retrouver `a` et `b`, puis vérifier que les tables retombent bien sur les mêmes abscisses.

```
p = 0xf6a443df32d5bcc4e9ea3d61f64521d067002154810ac3fbdf5b67c7d9be76d9
n = 0xf6a443df32d5bcc4e9ea3d61f64521d09fceed7d5af79c9a4f59ac517c85b37f
a = 0x5528912e3bce70014db3ef787435dd7417a37cf68830ea9299fd202b0489ff4f
b = 0xb41cee8f5aaacee166ed945ca7044a14dc432c639b8ed0cc690f83c7c175669e
```

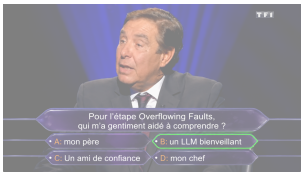
La cohérence des paramètres est vérifiée de deux manières : le hash de `a || b` correspond au contrôle embarqué, et les points `G`, `2G`, `3G` recalculés correspondent aux tables du binaire.
À ce stade on connaît aussi les deux clés importantes : l'ancienne clé légitime, dont on a récupéré la clé privée dans `prod_maj_key.sa`, et la clé publique SiviHaKerez, active sur `diode_dst`.

```
Q_old.x  = 0xc4d43632ce64e3cfe559e61d62859e0ec0660810e5fa35ad9b4596df33042341
d_old    = 0x9f1226a9113281ac0084f198baf5d6462499e684390b01fefe065a36af628dbe
Q_sivi.x = 0x9b3d009d95fcef43db6a31a95cc2a9f289afa1f78e9d6f3568f24dfc18b85bae
```

Ce flag intermédiaire valide surtout qu'on a les bons paramètres Lobster.

Flag Step 3.1

`SSTIC{94a19b2019010c12bc842074e0af93c0ba3a5be773ae7043fe891bbb408a261b}`



5 | Step 3 : overflowing faults

L'étape suivante est disponible sur :
http://51.15.164.185/step/df4bddd435bb9b4cf0896565705b9b3b
L'objectif est maintenant de faire accepter par `diode_dst` une archive dont le paquet contient `UTILS_GET_FLAG_STEP3`.
Côté source, on sait fabriquer une archive valide. Le problème est côté destination : la signature Lobster256 est vérifiée avec la clé publique SiviHaKerez, et on ne possède pas sa clé privée.

La vulnérabilité vient du décodage base64 de la signature. Une signature normale fait 64 octets `r || s`, encodés en 88 caractères avec `==`.
Mais si on fournit 88 caractères sans padding, le décodeur produit 66 octets. Le buffer destination fait 64 octets. Les deux octets en trop débordent au début de la structure de clé publique importée sur la pile.

Ces deux octets permettent de garder `key_type = 0x02`, puis de modifier l'octet faible de `X` : l'abscisse de la clé SiviHaKerez passe de `[...]85bae` à `[...]85b00`.

Ce point n'est alors plus situé sur la courbe elliptique attendue. Dans l'implémentation ANSSI, la multiplication scalaire passe par `prj_pt_mul`, qui vérifie que son entrée et sa sortie appartiennent à la courbe. Lobster remplace ici cette multiplication par `PRJ_XZ_ONLY_MUL` ; après l'overflow, la clé publique déjà importée n'est donc plus rejetée avant le calcul.
Le ladder x-only ne voit que l'abscisse : si celle-ci ne correspond plus à un point de la courbe principale, les mêmes formules décrivent le calcul sur le twist associé. Ce twist est exploitable ici car son ordre possède de petits facteurs, contrairement au grand sous-groupe prévu par la courbe originale. Cette factorisation rend le logarithme discret attaquable via Pohlig-Hellman sur plusieurs petits sous-groupes. On résout alors le DLP avec des sous-problèmes BSGS, ce qui donne un scalaire compatible avec la clé publique altérée telle qu'elle sera manipulée par l'implémentation.
La forge de signature consiste ensuite à revenir au schéma EC-KCDSA attendu par Lobster : choisir un nonce `k`, calculer le point `W = kG`, dériver `r` depuis l'abscisse de `W`, puis calculer `s` avec le scalaire récupéré. La signature `r || s` est encodée de manière à conserver les deux octets d'overflow `02 00` ; lors de la vérification, ces octets modifient la clé publique SiviHaKerez en la clé altérée pour laquelle la signature a été construite.

Appel à un ami forgeron

Pour cette étape, j'ai utilisé un LLM comme aide ponctuelle à la compréhension du mécanisme de twist et à l'écriture du code d'exploitation associé. Les résultats ont ensuite été vérifiés expérimentalement sur les binaires du docker fournis puis sur l'instance distante.

Une archive `GET_FLAG_STEP3` signée avec cette signature forgée passe localement puis sur le remote. VNC affiche enfin le flag dans les logs de `diode_dst`.

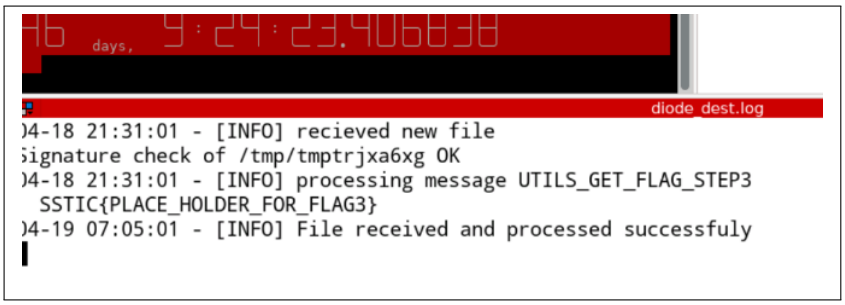


FIGURE 5.1 – Attaque sur le docker

Flag Step 3

`SSTIC{5579a85b0f2e9f87d6a4696b951d0dfcc6f2908e219a756e43e0b2e32112b397}`

6 | Step 4 : dancing in shadow

À ce stade, il est maintenant possible de réinstaller une clé de signature maîtrisée, ce qui évite de dépendre à chaque archive de la signature forgée de l'étape précédente.
http://51.15.164.185/step/48df3610c3412eb5f513de714fc28601?upgrade
Le step 3 donne un canal applicatif vers `diode_dst` : on dépose une archive `.sa` signée dans `/in`, et `diode_dest.py` exécute ses handlers. Ce niveau permet d'interagir avec `weapon_authent` au moyen des messages applicatifs destinés à `weapon_server`.

Le flux de commandes utilisé est :

```
WEAPON_OPEN_SESSION
WEAPONS_MSG
WEAPON_CLOSE_SESSION
```

En local, pour accélérer l'exploitation, j'ai exposé directement `weapon_authent` sur `1515`. Le binaire est un ELF x86_64 PIE stripped, supervisé par ShadowGuard.
Il charge `users_db.bin` au démarrage, puis échange avec `weapon_server` via deux pipes inter-processus.

La structure des messages applicatifs est la suivante :

```
u32be packet_size
u8    opcode
u16be error/status
u16be value_count
repeated TLV:
  u8 type
  u16be length
  bytes value
optional u16 CRC words
```

Les opcodes identifiés sont `AUTHENT`, `GET_TARGET`, `SET_TARGET`, `FIRE`, `DISARM`, `GET_VERSION` et `IMPERSONATE`. Après authentification avec `SSTIC_USER / DefaultPassword`, `GET_VERSION` renvoie bien :

```
0.1.1.3
```

Le bug se trouve dans le traitement des CRC optionnels placés après les TLV. Le parser accepte jusqu'à 64 entrées, puis traite autant de mots CRC de 16 bits. Ces mots sont combinés par XOR dans un buffer local de pile qui ne couvre que `0x48` octets. Or 64 mots de 16 bits représentent `0x80` octets : les premiers modifient bien la zone prévue, mais les suivants débordent sur les registres sauvegardés, puis sur l'adresse de retour.

```
rsp+0x00..0x47 : local CRC area
rsp+0x48       : saved rbx
rsp+0x50       : saved rbp
rsp+0x58       : saved r12
rsp+0x60       : saved r13
rsp+0x68       : saved r14
rsp+0x70       : saved r15
rsp+0x78       : saved return address
```

Comme le binaire est PIE, les adresses utiles dépendent de l'ASLR à chaque démarrage du processus. Pour éviter de recalculer la base à chaque tentative, je garde la même session ouverte : tant que `weapon_authent` n'est pas redémarré, les adresses restent stables.
Le premier paquet sert donc de probe. Il corrompt volontairement l'adresse de retour avec une valeur encore située dans le binaire. ShadowGuard détecte l'écart entre son adresse attendue et l'adresse réellement utilisée, puis l'affiche dans le monitoring VNC :

```
Shadow stack detection -> expected:0x5629c7c38484 got:0x5629c7c3719d
```

Le `got:` correspond à une adresse de retour corrompue mais encore située dans le binaire.
Dans le désassemblage de `weapon_authent`, cette adresse correspond à l'instruction `TEST EAX,EAX`, située à l'offset `0x219d`. Comme le binaire est compilé en PIE, il suffit de retrancher cet offset à l'adresse affichée par ShadowGuard pour retrouver la base de chargement :

```
PIE base = 0x5629c7c3719d - 0x219d = 0x5629c7c35000
```

Le paquet d'authentification précédent a laissé en pile un nom d'utilisateur contrôlé. Avec le retour forcé vers `PIE+0x219d`, le dispatcher reprend avec un `rsp` décalé : ses champs de requête pointent alors dans ce nom d'utilisateur.

```
username+0x10 : adresse à lire
username+0x18 : taille à lire
username+0x20 : opcode 5 (GET_VERSION)

ret forgé -> PIE+0x219d

226c: lea rbx, [rsp+0x60]
2271: mov esi, DWORD PTR [rsp+0x48]  ; taille
2275: mov rdi, QWORD PTR [rsp+0x40]  ; adresse
227a: mov rdx, rbx                   ; réponse
227d: call 1a20                      ; GET_VERSION
```

Ensuite on utilise le même canal pour lire les globals qui décrivent la base utilisateurs chargée par `weapon_authent`.
Le leak donne :

```
db_size = 0xfc68
db_ptr  = 0x5629f065a4b0
```

La même primitive est ensuite pointée sur `db_ptr`. Le `GET_VERSION` détourné sert alors de lecture mémoire : `weapon_authent` renvoie quelques octets pris à l'adresse demandée. `diode_dest.log` affiche cette réponse sous forme d'hexdump dans VNC. Comme le premier enregistrement de `users_db.bin` commence par le nom d'utilisateur, et que ce premier nom est le flag, lire le début de la base suffit à le faire apparaître.

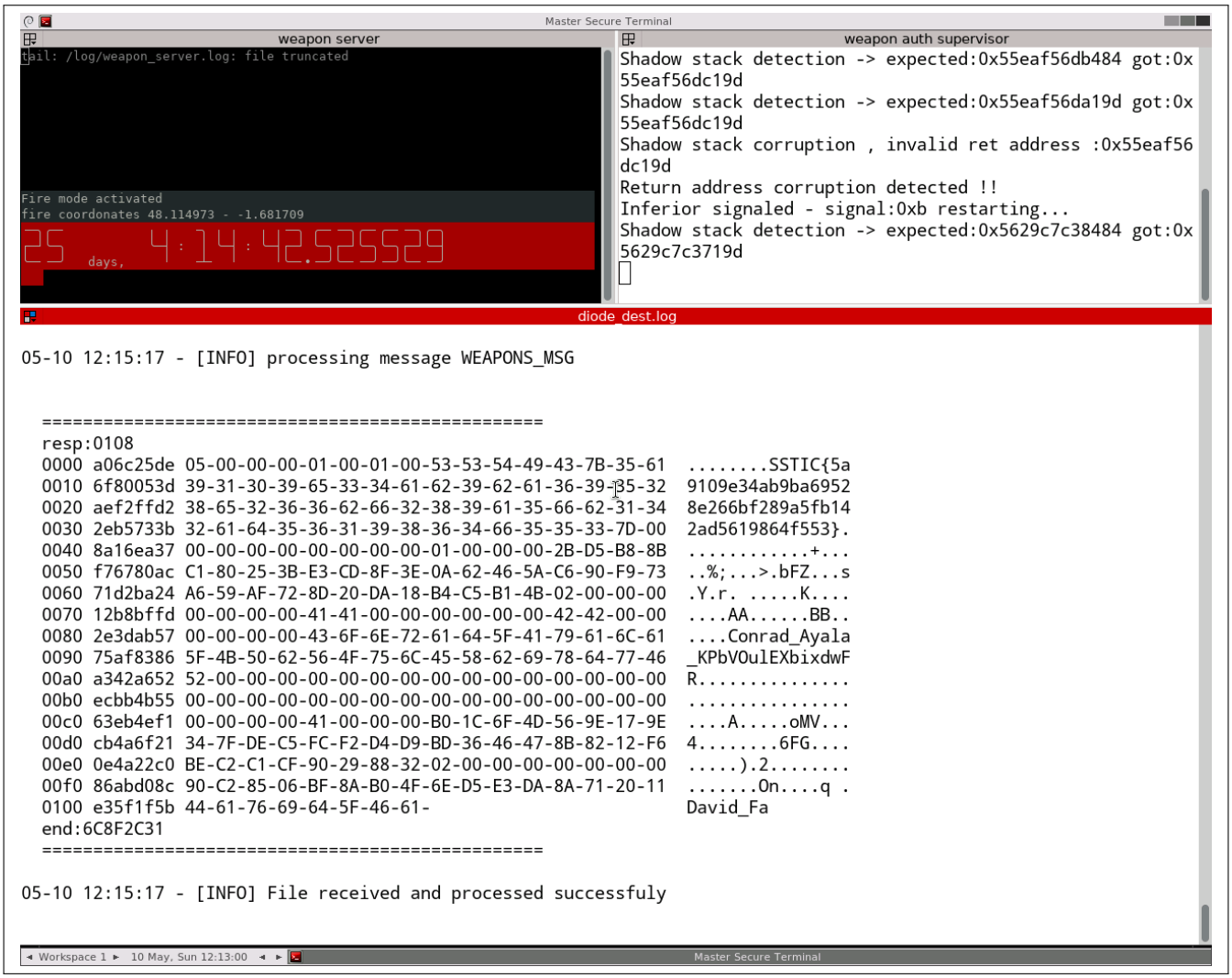


FIGURE 6.1 – Hexdump affiché dans VNC après lecture mémoire du début de `users_db.bin`.

Cette lecture suffit à valider *dancing in shadow*...mais...

Flag Step 4

`SSTIC{5a9109e34ab9ba69528e266bf289a5fb142ad5619864f553}`

P.S. J'ai essayé de condenser cette étape sur une seule page...mais j'ai pas réussi :]

7 | Step 5 bonus : dumping through my screen

Après la validation du step 4, le serveur renvoie vers fd0c9dd1f12907bf25f4fd7af0bd83e5.
Obtenir un flag, c'est bien ; éviter qu'une arme pointe vers le Couvent des Jacobins, c'est mieux.
Pour cela, il faut maintenant appeler la commande `DISARM`. Grâce au docker fourni et à son fichier `users_db.bin` de test, on comprend que la commande `IMPERSONATE` permet de passer d'un utilisateur à un autre, à condition qu'ils partagent au moins un groupe commun.
La structure de `users_db.bin` est la suivante :

```
struct user_record {
    char     username[0x40];
    uint32_t permissions;
    uint8_t  password_sha256[0x20];
    uint64_t group_count;
    uint64_t groups[group_count];
};
```

Le nom de l'étape bonus devient alors assez littéral : *dumping through my screen*. Il faut reconstruire l'intégralité de la base utilisateurs afin de trouver une chaîne d'impersonation permettant d'atteindre un compte possédant la permission `DISARM`.
On a une lecture mémoire, mais sur le remote elle passe par le flux applicatif puis par l'affichage VNC. Il faut transformer un hexdump affiché à l'écran en dump binaire de `users_db.bin`.
La méthode retenue consiste à envoyer un probe de lecture par chunk, capturer l'écran VNC avec `vncdotool -nocursor`, puis extraire les octets hexadécimaux affichés par le terminal. Les morceaux récupérés sont ensuite réassemblés pour reconstruire `users_db.bin`.

Une fois le dump assemblé, la base est parsée comme une liste d'enregistrements utilisateurs. Elle contient 476 comptes.
La relation de groupes permet ensuite de reconstruire une chaîne d'impersonation depuis le compte de départ vers le compte `audit_KaKaHuet`

```
[+] chain:
    0x48f8 perms=0x63 groups=(0x3ff8fe67586df890,0x159f3ede192587dc) SSTIC_USER [IMPERSONATE]
    0x3124 perms=0x41 groups=(0x1a520507b274f835,0x3ff8fe67586df890) Marvin_Thomas_BOHkZtJnLGdqHgtv [IMPERSONATE]
    0x49f8 perms=0x41 groups=(0x60262937fb212676,0x1a520507b274f835) Dalton_Zook_eWkgWqXCAWlxpwFu [IMPERSONATE]
    0x35d0 perms=0x41 groups=(0x60262937fb212676,0x215f1f524e5252b7) Andres_Carpenter_mOmguqWWFxMkkRqO [IMPERSONATE]
    0xac74 perms=0x19 groups=(0x215f1f524e5252b7,0x640de522365292cf) audit_KaKaHuet [DISARM]
```

On fabrique alors une archive normale qui authentifie `SSTIC_USER`, enchaîne les `IMPERSONATE`, puis envoie `DISARM`.

```
AUTH SSTIC_USER DefaultPassword
IMPERSONATE Marvin_Thomas_BOHkZtJnLGdqHgtv
IMPERSONATE Dalton_Zook_eWkgWqXCAWlxpwFu
IMPERSONATE Andres_Carpenter_mOmguqWWFxMkkRqO
IMPERSONATE audit_KaKaHuet
DISARM
```

Le serveur renvoie enfin l'adresse mail de fin :

Figure 7.1 – Adresse mail finale renvoyée par le serveur

Adresse finale

`777a6c006a0f848986e7420e3210640734535648ee507d0cc10d5d434314cc96@sstic.org`

Conclusion

Ce challenge SSTIC 2026 aura probablement été mon dernier.

La résolution complète m'a demandé un peu plus d'un mois de travail. Comme lors des deux précédentes éditions que j'ai réussi à valider, en 2019 et 2020.

En voyant certaines personnes résoudre l'ensemble en quelques jours, il devient assez évident que le niveau technique et l'investissement demandé dépassent largement ce que je peux raisonnablement consacrer à ce type de challenge aujourd'hui.

Même si l'expérience reste extrêmement intéressante techniquement, le coût en temps et en énergie finit par devenir difficile à justifier.

Cela reste malgré tout un très beau challenge, varié, cohérent, mais franchement brutal.

Je remercie également les organisateurs et concepteurs du challenge SSTIC 2026 pour le travail considérable derrière cette édition.

J'en profite aussi pour remercier mes deux petits monstres qui liront peut etre un jour ce document <3.