Analyse de sécurité de technologies propriétaires SCADA

Jean-Baptiste Bédrune, Alexandre Gazet et Florent Monjalet
jbbedrune@quarkslab.com
agazet@quarkslab.com
fmonjalet@quarkslab.com

Quarkslab

Résumé Les systèmes SCADA [1] sont au cœur de beaucoup de systèmes critiques, parmi eux : centrales nucléaires, circuits de distribution d'eau ou systèmes d'alarme.

Cet article relate l'étude de sécurité que nous avons menée sur des technologies SCADA récentes. Nous mettrons en particulier l'accent sur la méthodologie employée pour arriver à nos fins et les diverses méthodes utilisées : rétro conception en boîte noire, en boîte blanche et fuzzing. Nous aborderons à la fois la rétro-conception du protocole industriel, d'une partie de la pile d'un client du protocole et du firmware du PLC.

Cette étude a permis de mettre en évidence plusieurs failles dans les technologies concernées et de dévoiler une bonne partie du cryptosystème d'un protocole propriétaire. Nous présenterons d'ailleurs une attaque tirant parti d'une des failles mises en lumière.

Disclaimer : Les termes utilisés pour désigner les technologies sont volontairement imprécis. Nous sommes actuellement en contact avec le constructeur concerné pour corriger les vulnérabilités identifiées.

1 Introduction

1.1 Motivations

En écho à de récentes attaques sur des systèmes critiques (on citera Stuxnet[8] qui avait réussi à mettre hors service des centrifugeuses à uranium en Iran, ou plus récemment Havex, un RAT ciblant des systèmes SCADA), certains constructeurs ont fait, et font encore, des efforts très conséquents pour élever la sécurité de leurs produits au même niveau que celle des systèmes traditionnels.

Le lourd passé des systèmes SCADA en matière de sécurité rend l'analyse de ces nouveaux systèmes sécurisés d'autant plus intéressante.

1. Supervisory Control And Data Acquisition, voir section 1.2 pour plus d'informations.

1.2 Qu'est-ce qu'un système SCADA ?

Avant de rentrer dans le cœur du sujet, ce paragraphe présente rapidement ce que sont les systèmes SCADA et le rôle des composants que nous allons étudier dans ce document.

Définitions

L'acronyme SCADA désigne une sous-partie d'un tout plus grand, appelé « systèmes industriels » (ICS, pour *Industrial Control System* en anglais).

Le terme « systèmes industriels » désigne ici des systèmes informatiques s'interfaçant avec des systèmes physiques : vannes, moteurs, capteurs de températures, de pression... On retrouve ce type de système notamment dans les centrales nucléaires, les circuits de distribution d'eau, les systèmes d'alarme, de contrôle d'accès ou de vidéosurveillance et bien d'autres. En réalité, beaucoup de systèmes rentrent dans cette définition.

La partie SCADA désigne la partie supervision et contrôle du processus physique. Le dossier paru dans MISC 74 [6] présente ce type de systèmes bien plus en détail.

Composants

La partie SCADA d'un ICS est composée (principalement) de trois éléments :

Les automates industriels programmables (API, ou PLC en anglais, pour *Programmable Logic Controller*). C'est un équipement embarqué programmable transformant des entrées électriques en sorties électriques (par exemple ajuster une vitesse de rotation en fonction d'une température). Ces équipements sont typiquement équipés de processeurs ARM et peuvent éventuellement contenir un OS. La manière dont les sorties sont calculées à partir des entrées est définie par un programme utilisateur modifiable à volonté, c'est la grande différence avec un circuit électronique classique.

La station de programmation sert à concevoir et télécharger un programme utilisateur sur le PLC.

Les postes de supervision (aussi abrégés sous l'acronyme IHM pour Interface Homme Machine) ont pour but de superviser l'état du système physique et de déclencher des actions (fermer la vanne A, changer la vitesse des moteurs B et C...). C'est une abstraction du système physique. Cette supervision et ces actions se font via la

lecture et l'écriture de variables sur le PLC, grâce à un protocole dédié. Ces postes consistent souvent en une interface graphique sur laquelle apparaîtront des mesures, et où des boutons pourront déclencher des actions.

Particularités

Les équipements industriels, et particulièrement les PLCs, ont tendance à être très pérennes : jusqu'à une trentaine d'années de durée de vie pour certains. Ils sont également très coûteux et délicats à remplacer : les PLCs en eux-mêmes peuvent être onéreux, mais il faut aussi compter toute la phase de programmation et d'intégration d'un nouveau modèle dans un système existant. L'une des conséquences de ce phénomène est de retrouver beaucoup de vieux équipements dans les milieux industriels.

Lorsqu'ils sont en fonctionnement, certains systèmes industriels (notamment les systèmes critiques) sont délicats à mettre à jour. Lorsqu'il n'y a pas une redondance complète du système, il faut pouvoir l'arrêter le temps de la mise à jour. Lorsque le degré de redondance du système permet d'en retirer des éléments pour les mettre à jour sans l'arrêter, il faut toujours s'assurer que la mise à jour ne nuit pas au système. Réaliser une simple mise à jour sur ce type de système est souvent un problème complexe, il n'est par conséquent pas rare de rencontrer des systèmes industriels ne bénéficiant pas des derniers correctifs de sécurité.

Du fait de leur rôle critique, même un déni de service sur un PLC est une situation qu'il faut à tout prix éviter. En règle générale, on cherchera à éviter toute situation modifiant le comportement de l'appareil ; les notions d'authenticité et d'intégrité sont souvent plus importantes que la confidentialité des données ou des communications dans ce genre de réseau. Ces exigences en matière de stabilité et de sécurité contrastent avec la réalité du terrain : ce sont souvent des équipements anciens et peu à jour. Le lecteur saisira donc toute l'importance de concevoir des équipements stables et proposant des mécanismes de sécurité robustes.

1.3 Les équipements étudiés

Nous avons choisi de nous concentrer sur une grande marque d'équipements industriels, sélectionnée essentiellement en raison de sa grande part de marché en Europe et de ses efforts pour améliorer la sécurité de ses équipements.

Nous avons étudié un de leurs PLCs récents, leur logiciel de supervision (appelé IHM dans la suite) ainsi que deux versions d'un de leurs protocoles propriétaires. La figure 1 montre comment ces éléments sont agencés.

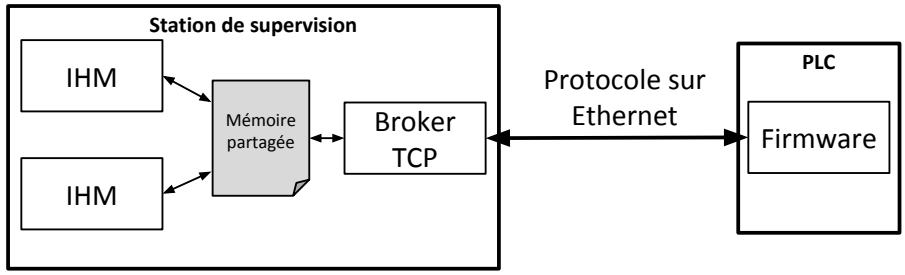

Figure 1. Architecture de travail

Une première cartographie du PLC révèle qu'il n'y a que deux services accessibles : un serveur web (en HTTP et en HTTPS, configurable et désactivable via le logiciel de programmation de PLC) et un serveur TCP pour le protocole métier. Ce type de protocole permet à la supervision (client du protocole) et au PLC (serveur du protocole) d'échanger des informations, ainsi que de reprogrammer le PLC. Ils peuvent être transportés sur Ethernet via TCP/IP, mais aussi sur diverses liaisons série.

Si de nombreuses vulnérabilités concernant les serveurs Web embarqués de PLCs ont été publiées, très peu concernent directement les protocoles métier, et encore moins lorsqu'ils sont propriétaires. L'avantage de s'intéresser directement au protocole métier est qu'il est très improbable de fermer le serveur de communications métier : sans cela, le PLC devient beaucoup moins utile. Le serveur Web est en revanche totalement optionnel (bien que pratique) et peut être désactivé. Il est d'ailleurs désactivé par défaut.

Nous avons donc choisi de nous intéresser au protocole métier, ici porté sur TCP/IP. Il en existe deux versions : la plus ancienne ne comportait pas (ou peu) de mécanismes de sécurité et est assez bien connue de la communauté. La seconde, sur laquelle nous avons concentré plus d'efforts, implémente un certain nombre de mécanismes de sécurité, mais il n'existe que très peu de travaux publics dessus.

2 Fuzzing d'un protocole connu

Afin de nous familiariser avec l'environnement SCADA du constructeur que nous avons choisi, nous avons décidé de commencer par implémenter une partie du protocole le moins récent, puis de fuzzer l'IHM avec en écrivant un « faux » PLC qui communique avec ce protocole. Celui-ci a

l'intérêt d'être relativement bien connu de la communauté, ce qui facilite grandement son implémentation.

2.1 Architecture de fuzzing

Comme dit plus haut, le but est de se faire passer pour un PLC auprès de la supervision, puis de la fuzzer, c'est-à-dire de lui envoyer des paquets malformés en espérant provoquer un comportement imprévu.

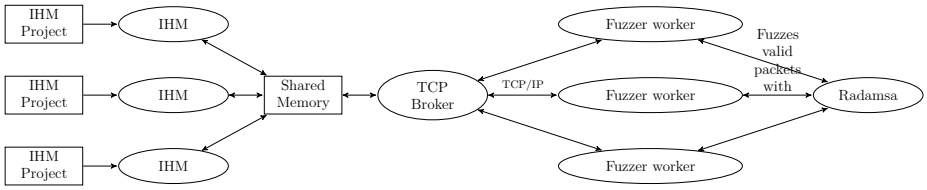

Figure 2. Architecture de fuzzing

L'architecture de fuzzing en elle-même (présentée en figure 2) reste assez classique :

- Plusieurs processus à fuzzer s'exécutent sur une VM (les IHMs).
- Plusieurs workers d'un même fuzzer s'exécutent sur une autre VM.
- L'IHM étant le client de la communication, c'est elle qui se connectera au fuzzer en configurant le fichier projet de l'IHM correctement.
- Le fuzzer est une implémentation d'un client légitime en Scapy, dont les payloads sont fuzzés par `radamsa`[1].

En revanche, il y a une particularité : les IHMs ne communiquent pas directement avec le fuzzer. Si cela n'empêche pas d'analyser les crashes produits, il est beaucoup plus compliqué de retrouver la capture réseau qui en a causé un : le `TCP Broker` n'offre pas de moyen documenté de savoir quelle instance d'IHM utilise quelle connexion TCP.

Pour résoudre ce problème, il faut arriver à associer une communication réseau à un PID d'IHM. Il n'est pas possible d'agir sur le port TCP de destination de la communication, et le port source ne donne pas d'indication sur le PID qui utilise la connexion TCP (à cause du *Broker*).

Une astuce à base d'IP aliasing a été utilisée : chaque projet est configuré pour se connecter à une adresse IP différente [2], et les workers

2. Il est possible de retrouver l'offset de l'adresse IP du PLC dans un fichier projet assez facilement.

du fuzzers sont configurés pour écouter sur toutes ces adresses IP. Ainsi, le script lançant les IHMs du côté client est capable d'associer le PID d'un processus ayant crashé avec l'adresse IP du projet qui lui a été passé en argument, et demande au fuzzer la dernière capture associée à cette adresse IP. Cette capture est nécessairement celle qui a fait crasher le processus côté client, puisque toutes les IHMs lancées à un moment donné se connectent à des IP différentes.

Ainsi, nous avons pu mettre en place une architecture de fuzzing où les ressources de chaque VM sont utilisées au maximum, et où il est possible de lancer k VMs contenant i IHMs et n VMs faisant tourner chacune un fuzzer à j workers, pour tous k, n, i et j tels que $k * i = n * j$.

2.2 Résultats

Après très peu de temps (environ 5 minutes de fuzzing sur un seul processus), le premier crash a eu lieu.

Il s'agit essentiellement d'une mauvaise vérification sur une valeur permettant d'accéder à un indice arbitraire dans une table de handlers. Le crash se produit lors de la phase de négociation du protocole, sur les premiers paquets.

L'exploitabilité de ce bug n'a pas été confirmée, mais il présente au moins l'intérêt de pouvoir faire crasher une IHM en 3 paquets, sans authentification nécessaire. Dans ce protocole, la supervision joue le rôle de client, et le PLC de serveur. Pour réaliser cette attaque, il faut arriver à intercepter un établissement de connexion TCP, puis répondre à la place du PLC avec les paquets frauduleux. Cependant, une autre manipulation permet de facilement faire un *reset* de la connexion TCP, facilitant beaucoup les choses.

3 Rétro conception d'un protocole industriel récent

Une fois l'architecture prise en main, nous avons choisi de commencer l'étude de la nouvelle version du protocole. Celle-ci propose notamment une authentification par mot de passe, donnant accès à différents niveaux de privilèges sur le PLC (grossièrement lecture et écriture). Très peu de travaux publics existent sur ce protocole ; or, il nous semble intéressant de vérifier qu'il est aussi sécurisé qu'il souhaite l'être.

3.1 Première approche : analyse en boîte noire

Le protocole que nous étudions n'est pas basé sur du texte. Face à un protocole presque inconnu et plusieurs dizaines de mégaoctets de

DLLs, nous avons choisi de commencer par une analyse en boîte noire. En pratique, il s'agit de regarder le contenu de captures réseau générées astucieusement en utilisant le logiciel d'IHM et le PLC du constructeur.

L'approche que nous avons employée est celle d'une analyse par différence (inspirée par [7] et [2]). Grâce à un outil permettant d'aligner les hexdumps des paquets (`hexlighter` [4]) et de mettre en surbrillance les différences entre deux paquets consécutifs, on en apprend beaucoup sur la structure du protocole.

Prenons un exemple de jeu de test où la supervision est censée lire tous les bits d'entrée et de sortie du PLC, pendant que nous les faisons varier en changeant les tensions d'entrée. Nous nous intéresserons ici uniquement aux réponses du PLC à la supervision.

Hexlighter permet (entre autres) de colorer des hexdumps de manière à faire apparaître les différences entre eux. Le résultat étant peu lisible sur les pages de ce livre, une représentation graphique remplaçant chaque octet par un carré de couleur a été proposée. La figure 3) montre le résultat en filtrant uniquement les réponses venant du PLC :

— Chaque ligne représente un payload de paquet (d'environ 110 octets ici).
— Un carré blanc est représente un octet identique à la ligne (au paquet) précédente.
— Un carré noir représente une absence de valeur (par exemple lorsqu'une ligne est plus courte). Il n'y en a pas dans cet exemple.
— Un carré vert (gris pour la version papier) indique une différence avec la ligne (le paquet) précédente.
— Plus le vert est vif (gris clair en version papier), plus la différence est importante (en valeur absolue).
— Dans la figure du bas, chaque ligne est comparée à la première plutôt qu'à la précédente.

Avec cette représentation, plusieurs choses sautent aux yeux :

— Un état stable (la forme du paquet ne varie plus) est atteint à partir du quatrième paquet envoyé (ces paquets ne sont pas représentés sur la figure, par souci de clarté). On en déduit donc que les trois premiers paquets doivent faire partie d'une sorte de handshake, et les autres sont le corps de la communication. Laissons le handshake de côté pour le moment.
— Un numéro de séquence apparaît clairement à l'offset 12 de tous les paquets (la taille de ce champ sera vérifiée en laissant le numéro de séquence augmenter jusqu'à ce qu'il repasse à zéro). Sur la figure 3, cela se traduit par une ligne verticale à l'offset 12 (légèrement

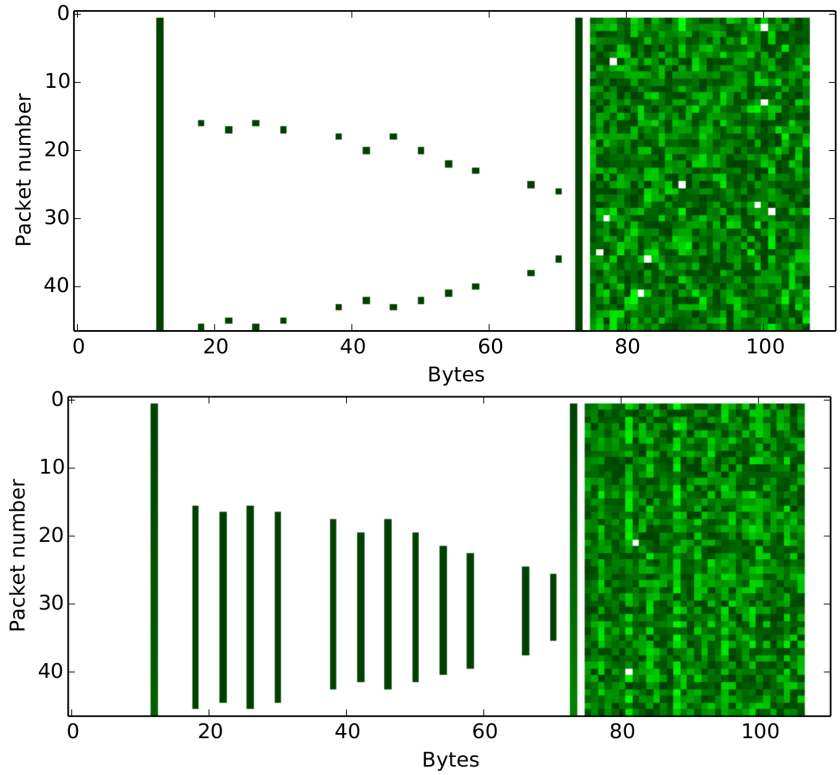

Figure 3. Représentation des différences entre paquets successifs similaires.
Chaque ligne représente un paquet. En haut, différence avec le paquet précédent, en bas la différence avec le premier paquet.

dégradée sur la figure du bas, car la différence avec le numéro de séquence du premier paquet augmente avec le temps).

— Un second numéro de séquence apparaît à l'offset 73. Celui-ci augmente de deux en deux, d'où le dégradé plus prononcé sur la figure du bas.

— À la fin de chaque paquet, un champ à forte entropie de 32 octets diffère beaucoup pour tous les paquets. Ce champ fait *très fortement* penser à de la cryptographie, par exemple un HMAC ou un hash.

— Un octet (en réalité un bit) change au milieu de certains paquets. Ces changements sont cohérents avec les variations d'I/O appliquées au PLC pendant la capture, nous en déduisons que ces champs représentent les variables du PLC lues par la supervision. Les lignes

vertes au milieu des paquets sur la figure du bas représentent le laps du temps où une variable à une valeur différente de sa valeur au premier paquet.

En y regardant de plus près et en comparant les paquets de manière différente (par exemple le cinquième paquet de plusieurs connexions différentes), on peut déduire le sens de nombreux champs. Chaque changement de paramètre (écriture au lieu de lecture, changement de valeur des variables, nouvelle session...) a tendance à révéler son impact dans les *diffs* de paquets. La clé du succès de cette méthode est toujours de pouvoir facilement associer un changement de paramètre à sa répercussion dans le trafic réseau.

3.2 Analyse en boîte blanche

L'analyse en boîte noire a ses limites. Après avoir mis en évidence beaucoup de champs intéressants du protocole, il reste certains problèmes qu'il n'est pas facile de résoudre sans une phase de rétro conception sur les binaires des clients implémentant le protocole. C'est notamment le cas du champ de 32 octets à forte entropie (à ce stade, un fort soupçon de HMAC ou de hash pèse sur cette partie du paquet), dont nous voulons connaître la signification précise.

La solution de facilité est de regarder le logiciel de supervision, car il s'exécute sur un ordinateur de bureau classique, en x86. Contrairement au PLC, son instrumentation est triviale.

Le premier problème auquel on se heurte est celui de la complexité de ce programme : plusieurs processus communiquent au travers de pages mémoires partagées, tous multithreadés, asynchrones et flanqués de plusieurs dizaines de DLLs, aux noms pas toujours très évocateurs.

Approche par trace de données

La première approche, naïve, a été de tracer la remontée des données depuis l'appel système `recv` jusqu'à la vérification du champ qui nous intéresse. Les nombreuses copies du buffer lu, sa migration entre les processus via une mémoire partagée et leur nature asynchrone ne facilitent ni l'analyse statique, ni l'analyse dynamique. La conclusion est simple : nous avons mis un terme à cette traque avant qu'elle ne porte ses fruits. En revanche, elle a permis de découvrir beaucoup de code intéressant et d'identifier le processus faisant réellement la vérification de ce champ, mais pas la fonction précise appelée.

Une approche intéressante serait de travailler sur une trace complète du programme, puis de tracer la destination des données avec du tainting.

Ce n'est cependant pas l'approche que nous avons utilisée, notamment à cause de sa difficulté de mise en place sur ce type de programme.

Approche par détection d'algorithme cryptographique

En réalité, nous avons une vague idée de ce que nous cherchons : les 32 octets qui nous intéressent impliquent assez probablement un hash. On peut imaginer qu'il s'agit soit d'un hash pour assurer l'intégrité du message (mais une taille de 32 octets semblerait disproportionnée pour cela), soit un HMAC ou équivalent. Si une implémentation d'un algorithme de hash produisant 32 octets (par exemple SHA 256) était présente dans une dll utilisée par l'IHM, il ne serait pas très coûteux de vérifier s'il est impliqué dans la génération de ces données ou non.

La méthode classique pour retrouver ce genre d'algorithme est de rechercher des constantes qui les caractérisent dans les binaires. La méthode employée consiste donc à lancer `signsrch`[5] sur toutes les DLLs utilisées par l'IHM lorsqu'elle est connectée au PLC, afin trouver une ou plusieurs DLLs présentant des algorithmes intéressants. Sur l'une d'elles (appelons-la `hmi_core.dll`), `signsrch` produit une sortie particulièrement intéressante :

```
offset   num  description [bits.endian.size]
------------------------------------------
xxxxxxxx 1036 SHA1 / SHA0 / RIPEMD-160 initialization [32.le
    .20&]
xxxxxxxx 2053 RIPEMD-128 InitState [32.le.16&]
xxxxxxxx 876  SHA256 Initial hash value H (0x6a09e667UL) [32.
    le.32&]
xxxxxxxx 1016 MD4 digest [32.le.24&]
xxxxxxxx 1299 classical random incrementer 0x343FD 0x269EC3
    [32.le.8&]
[...]
xxxxxxxx 1290 __popcount_tab (compression?) [..256]
xxxxxxxx 874  SHA256 Hash constant words K (0x428a2f98) [32.le
    .256]
xxxxxxxx 894  AES Rijndael S / ARIA S1 [..256]
xxxxxxxx 897  Rijndael Te0 (0xc66363a5U) [32.be.1024]
xxxxxxxx 899  Rijndael Te1 (0xa5c66363U) [32.be.1024]
xxxxxxxx 901  Rijndael Te2 (0x63a5c663U) [32.be.1024]
xxxxxxxx 903  Rijndael Te3 (0x6363a5c6U) [32.be.1024]
xxxxxxxx 915  Rijndael rcon [32.be.40]
[...]

- 18 signatures found in the file in 7 seconds
```

On y retrouve le SHA 256 suspecté, ainsi que d'autres algorithmes de hash et de chiffrement (dont AES, qui nous intéressera plus tard).

En plaçant un *breakpoint* sur la partie `sha2_process` de SHA 256 dans `hmi_core.dll`, on peut vérifier que cette fonction est bien appelée

pendant la communication, et semble être appelée plusieurs fois à chaque paquet reçu ou envoyé.

Un peu d'analyse statique permet de déterminer que les fonctions de SHA 256 servent à réaliser un HMAC. En déboguant le programme, on peut bien constater que le payload du paquet est passé en paramètre au HMAC avec une sorte de clé de session de 24 octets. La sortie du HMAC SHA 256 correspond bien aux 32 octets de forte entropie à la fin du paquet dont le payload a été passé en paramètre au HMAC.

Cette méthode peut permettre d'assurer l'authenticité de chaque paquet émis, dans la mesure où la clé est unique à la session et échangée de manière confidentielle. C'est une belle avancée, mais une question importante est maintenant en suspens. D'où vient cette clé, et comment est-elle échangée ?

3.3 Découverte du cryptosystème

Génération de la clé de session

Il se trouve que tracer la génération de la clé de session a été beaucoup plus facile, ce ne fut l'affaire que de quelques *breakpoints* en écriture sur les données intéressantes.

L'identification de l'origine de la clé de session, générée par un PRNG, a permis de mettre en évidence un premier problème : la sortie du PRNG n'a rien d'aléatoire. En effet, aucun aléa ne lui est fourni (il est toujours réinitialisé de la même façon, avec un buffer constant). Ceci est confirmé par le fait que la séquence de clé de sessions générée est toujours la même lors d'une exécution du logiciel de supervision. Il est ainsi possible de bruteforcer cette clé en un temps très court (en générant la même séquence de clé que la supervision), puis de voler une session déjà authentifiée en outrepassant l'authentification par mot de passe normalement proposée par le protocole. Une attaque exploitant cette vulnérabilité sera présentée dans la section 4.

Échange de la clé de session

Cette attaque sur l'usage biaisé du PRNG est intéressante, mais il serait dommage de s'arrêter à cette étape : il s'agit d'une erreur d'implémentation relativement facile à corriger. Nous cherchons maintenant à savoir comment l'IHM fournit cette clé de session au PLC.

Un rapide retour à de l'analyse en boîte noire nous permet d'identifier le paquet portant a priori la clé de session. Le raisonnement est le suivant :

— Le premier paquet envoyé par la supervision est toujours le même (à un numéro de séquence près) et est envoyé avant la génération de la clé de session.
— La réponse du PLC est constante à l'exception de 3 octets épars et un bloc de 20 octets qui diffèrent.
— Le deuxième paquet de la supervision varie toujours de manière significative d'une session à une autre. Un bloc de 132 octets est le plus intéressant, car il s'agit du seul qui soit assez grand pour pouvoir transmettre la clé de session (chiffrée ou encodée).
— La réponse du serveur s'avère être presque vide (accusé de réception applicatif).
— Le troisième paquet envoyé par le client est déjà authentifié avec un HMAC.
— Il en est de même pour le troisième envoyé par le serveur.

Il semble donc évident que le passage de la clé de session se fait dans le deuxième paquet envoyé par le client au serveur.

Rappelons qu'une implémentation d'AES est présente dans la bibliothèque `hmi_core.dll`. Aucun algorithme de chiffrement asymétrique n'étant détecté dans la dll (à première vue), AES devient un bon candidat pour ce qui est de l'échange d'un secret. En déboguant le processus de l'IHM, on peut constater que les fonctions utilisées par AES sont bien appelées au moment de forger le paquet qui nous intéresse.

Après une analyse statique de cette implémentation, nous avons pu reconstituer l'algorithme complet qui est une forme d'AES 128 en mode GCM, qui est très proche du mode CTR. Pour rappel, le principe du mode CTR est de *xor*-er le texte clair avec un *keystream*. Ce *keystream* est généré en dérivant l'IV pour obtenir une suite d'octets de longueur arbitraire et en chiffrant chacun de ces IVs dérivés par bloc de 128 bits (pour AES 128).

En connaissant l'algorithme et la clé (elle peut être récupérée en débuggant l'IHM pour une session donnée), nous pouvons maintenant déchiffrer la clé de session utilisée pour une communication donnée. Nous vérifions ainsi que la clé de session est bien chiffrée avec AES 128 GCM (avec quelques autres données) et que cela représente les 72 derniers octets du buffer de 132 octets précédemment identifié.

Il se trouve cependant que cette clé AES change lors de chaque session (bien que la séquence générée lors d'une exécution du processus soit toujours la même, à cause du PRNG biaisé). Nous repartons donc pour une nouvelle itération : comment cette clé AES est-elle générée et communiquée au PLC ?

Échange d'un premier secret partagé

Résumons la situation :

— Le client du protocole est authentifié grâce à un mot de passe (le mécanisme ne sera pas présenté dans cet article, mais est ici considéré comme fiable).
— Un HMAC des paquets avec une clé de session secrète garantit que le client qui s'est authentifié est bien celui qui envoie des paquets.
— Cette clé de session est générée côté supervision, puis chiffrée avec AES et envoyée au PLC.
— Comment le PLC a-t-il connaissance de la clé AES qui lui permet de déchiffrer la clé de session ?

Nous réitérons avec la même méthode que précédemment : équipés des dernières informations obtenues, un peu d'analyse en boîte nous permet d'isoler la partie du protocole qui pourrait être responsable de cet échange : il s'agit des 60 premiers octets du buffer de 132 octets précédemment mentionné.

En corrélant ces données avec celles traitées par le processus en amont du chiffrement AES précédemment identifié, nous retrouvons une partie du code qui semble « chiffrer » la clé.

À notre grande surprise, le code réalisant cette opération semble obfusqué, et à l'heure où ces lignes sont écrites, nous n'avons pas pu identifier l'algorithme utilisé.

Conclusion sur le cryptosystème

La figure 4 résume le système cryptographique que nous avons découvert. La fonction *obf_enc* représente le chiffrement obfusqué que nous n'avons pas identifié. Le mode d'authentification par mot de passe n'a pas été examiné au moment de la rédaction de cet article.

Les briques du cryptosystème que nous avons pu identifier semblent standard et solides. Cependant, plus de temps sera nécessaire pour nous prononcer sur la solidité du système complet et en particulier de l'algorithme obfusqué. Il reste encore des zones d'ombres dans le système cryptographique et la façon dont les données sont échangées peut laisser penser que ces algorithmes s'appuient sur des secrets fixés dans le firmware ou le hardware du PLC. Ces secrets pourraient être communs à tous les PLCs d'un même modèle et ayant la même version de firmware.

À ce stade, il nous a semblé intéressant d'aller voir du côté du firmware, notamment pour vérifier si le pendant de la routine à identifier était également obfusqué côté PLC. Cet équipement ayant des ressources plus

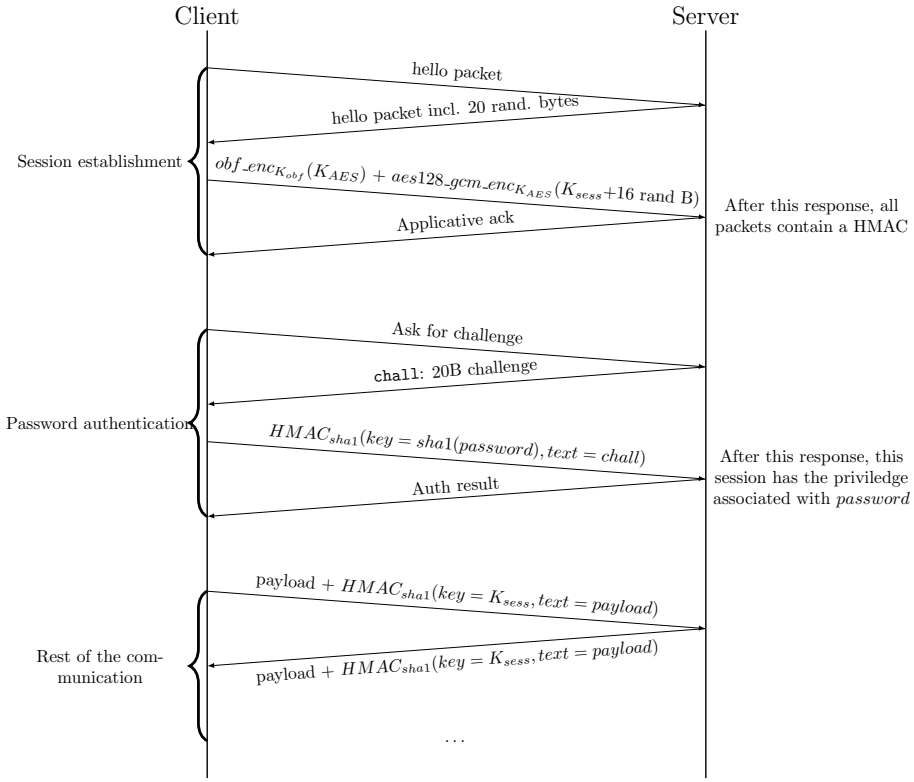

Figure 4. Résumé du système cryptographique

limitées, il ne serait pas étonnant d'avoir (au moins) une obfuscation plus légère.

Dans le cas où le code du côté PLC serait plus simple, cela permettrait sûrement d'identifier plus facilement l'algorithme utilisé. C'est ce que nous traitons dans la section 5. Mais avant de présenter cette étude, nous décrirons l'attaque permettant de voler une session authentifiée de ce protocole.

4 Man-in-the-middle avec vol de session authentifiée

Comme montré dans la section 3.3, un manque d'entropie en entrée du PRNG utilisé rend ses sorties prévisibles : il s'agit toujours de la même séquence, à chaque exécution du client. Cette partie décrit les étapes qui ont permis d'écrire un code démontrant la possibilité de cette attaque.

4.1 Résumé du problème

Toute la sécurité d'une session repose sur le fait que seuls le client et le serveur connaissent un secret partagé, que nous appelons clé de session. Il se trouve que la suite des valeurs retournée par le PRNG est parfaitement prévisible, en raison d'un défaut d'usage : le PRNG est toujours initialisé avec des chaînes constantes. Il est par conséquent possible d'énumérer toutes les clés de session dans l'ordre où elles sont générées.

4.2 L'attaque

Description

À partir d'un seul paquet authentifié, il est possible de vérifier si une clé de session donnée a été utilisée pour générer le HMAC authentifiant le paquet. Étant donné qu'il est possible d'énumérer les clés de session générées par l'IHM dans l'ordre, une attaque consistant à générer puis tester les clés une par une prend un temps très raisonnable. Cette attaque permet de retrouver la clé de session qui a servi à authentifier le paquet capturé ; cette clé de session permettra d'authentifier n'importe quel autre paquet, permettant de réaliser des actions avec les privilèges de la session volée.

Le premier objectif sera de modifier les réponses aux requêtes de lecture du client pour modifier l'état qu'il perçoit du PLC. Le second sera d'écrire des valeurs arbitraires sur le PLC avec les droits de la session sans que celle-ci s'en aperçoive. La figure 5 montre les LEDs indiquant les entrées (LEDs du haut) et sorties (LEDs du bas) du PLC, ainsi qu'une portion de l'interface de supervision (haut de la figure). On note que les pastilles vertes et rouges de la supervision représentent les LEDs du PLC, et que leurs états sont synchronisés.

Cette attaque nécessite d'avoir un accès en lecture au réseau. Il est envisageable d'arriver à des résultats intéressants sans man-in-the-middle, à condition de pouvoir au moins capturer un paquet authentifié, bien que cet angle d'attaque ne soit pas étudié ici.

Étape 1 : Man-in-the-middle TCP

Afin de pouvoir intercepter et modifier les paquets échangés par le client et le serveur via TCP, la première étape consiste à faire transiter tous ces paquets par la station de l'attaquant. En pratique, cela peut être par exemple réalisé avec de l'ARP spoofing.

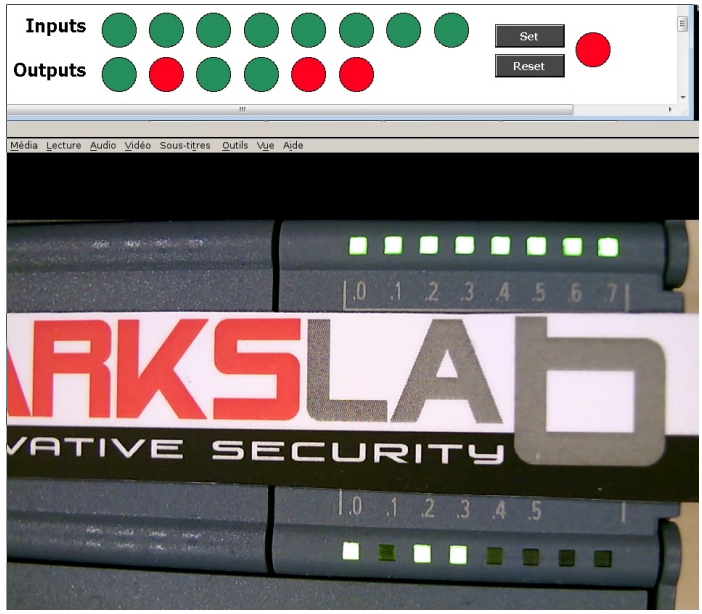

Figure 5. Aperçu des entrées et sorties du PLC et d'une partie de l'interface de supervision (en haut)

Le man-in-the-middle est basé sur l'usage des NFQueues pour faire remonter les paquets en espace utilisateur, les modifier et les renvoyer. La commande pour envoyer les paquets routés dans une nfqueue est :

```
sudo iptables -I FORWARD -j NFQUEUE --queue-num 1 [<various
    filters>]
```

Un binding python pour l'API de nfqueue a été utilisé. Ce binding n'implémente pas la fonction `set_payload`, pour pouvoir quand même réaliser l'attaque, les paquets modifiés sont *droppés* par la nfqueue et renvoyés en *raw socket* après modification.

Comme nous le verrons dans la suite, certaines modifications de paquets changent également leur taille, désynchronisant les `seq` et `ack` des deux parties de la connexion TCP. Ne s'agissant que d'une preuve de concept, la gestion de cette désynchronisation dans le man-in-the-middle n'a pas été implémentée, causant le *reset* de la connexion TCP dans certains cas.

Étape 2 : Retrouver la clé d'une session

Cette partie est relativement simple : il suffit d'isoler le HMAC (le bloc de 32 octets de forte entropie vu précédemment) d'un paquet, de

générer toutes les clés de session dans l'ordre où l'IHM les génère et de les tester une par une. Le nombre de sessions établies par l'IHM depuis son démarrage étant raisonnablement faible, cette attaque aboutit très vite.

Étape 3 : Authentifier des paquets arbitraires

Connaissant la clé de session, il est possible d'authentifier des paquets arbitraires. Le plus simple est de partir d'un paquet légitime, de le modifier, puis de l'authentifier en recalculant le HMAC. Une méthode de vérification est de relancer le bruteforce sur le nouveau paquet authentifié et de vérifier que la clé retrouvée est bien celle utilisée pour authentifier le paquet.

Étape 4 : Modification des réponses aux requêtes de lecture

Pour simplifier, nous construirons l'attaque en fonction d'un projet d'IHM en particulier. Il est possible de faire quelque chose de plus générique, mais cela demande plus de temps et n'apporte rien à la preuve de concept.

Pour un projet donné, les requêtes de lecture et les réponses associées ont toujours le même payload. Pour contrôler les valeurs reçues par la supervision (le client), il suffit de repérer, dans le paquet de réponse les offsets des valeurs des variables du PLC, puis de les mettre à la valeur voulue. On rajoute ensuite les données d'authentification avec la méthode décrite précédemment, puis on envoie le paquet à la supervision.

Cette méthode permet d'atteindre le premier but : contrôler ce que reçoit la supervision, et donc ce que percevrait un opérateur sur une IHM dans un vrai site industriel. Un exemple de mise en oeuvre de cette attaque est visible sur la figure 6 : l'état de l'interface de supervision (en haut) n'est plus cohérent avec l'état réel des LEDs du PLC.

Étape 5 : Génération d'une requête d'écriture Cette partie est un peu plus complexe. Pour la mener à bien, le plus simple est de substituer une requête de lecture par une requête d'écriture, qui devra être presque intégralement forgée. Se substituer à une requête existante évite de devoir forger la partie TCP du paquet.

Il s'agit ensuite de générer une requête d'écriture qui soit bien acceptée par le PLC. Pour cela, nous partons d'une requête d'écriture légitime générée par le logiciel de supervision et préalablement capturée. Pour déterminer quels champs changer pour arriver à une requête valide dans le contexte d'une session quelconque, il suffit de regarder les différences entre deux paquets d'écriture légitimes ou entre un paquet forgé invalide et un paquet valide. La plupart de ces champs sont simples à reconstruire,

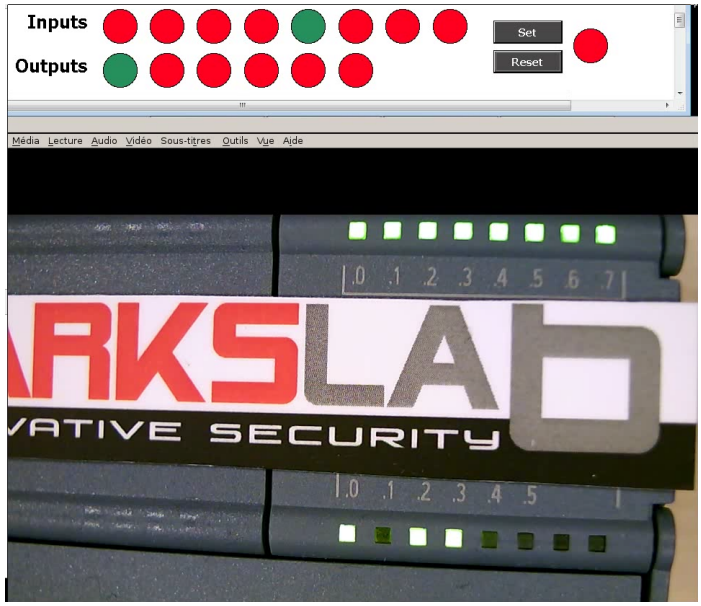

Figure 6. Désynchronisation entre les LEDs d'un PLC et l'interface de supervision

car ils dépendent soit du payload du paquet courant, soit du précédent paquet envoyé par la supervision, quel qu'il soit.

Un champ, en revanche, est plus complexe à forger : il s'agit d'un numéro de séquence propre aux requêtes d'écritures. S'il n'est pas cohérent, le PLC refusera la requête. Pour récupérer ce numéro de séquence, il y a trois méthodes :

— Avoir la chance d'intercepter une requête d'écriture faite par l'IHM. Toutes les IHM ne font pas forcément de requêtes d'écriture, il n'est donc pas toujours possible de compter sur ce scénario.
— Faire un reset de la connexion : cela force la supervision à refaire une authentification par mot de passe. Lors de celle-ci, une requête de type *write* est envoyée par la supervision, permettant de récupérer le dernier numéro de séquence d'écriture valide. S'il n'y a pas d'authentification par mot de passe, alors l'attaque n'a pas d'intérêt.
— Bruteforcer le numéro de séquence jusqu'à ce que le PLC accepte la requête d'écriture. Cette solution est moins viable, et dans certains cas moins discrète.

Par un procédé d'essai-erreur itératif, nous avons fini par réussir à générer des requêtes d'écriture valides pour une variable arbitraire

d'un projet donné (seule la modification des variables de sortie a été implémentée).

Conséquences

Si cet exemple se cantonne aux sessions de supervision, il peut aussi s'étendre aux sessions d'administration, permettant par exemple à un attaquant de voler une session de reprogrammation de PLC pour télécharger un programme arbitraire. Toute communication établie par un client vulnérable utilisant ce protocole peut être compromise par cette attaque.

Un correctif pour cette vulnérabilité a depuis été proposé par le constructeur.

Cette attaque est relativement réaliste : en pratique, il est souvent possible d'arriver à accéder à un réseau industriel. Certains d'entre eux sont connectés à des réseaux d'entreprise, voir directement à internet. Quand ce n'est pas le cas, ils utilisent parfois des réseaux sans fil (WiFi, GSM) pour fonctionner. Même lorsque ces vecteurs d'attaque ne sont pas disponibles, il y aura souvent un moment où un ordinateur portable d'un agent de maintenance finira par être relié au réseau industriel, rendant possible cette attaque. Les réseaux étant souvent "à plat", sans routage, cela rend l'attaque d'autant plus réalisable.

5 Accès au firmware

Les firmwares sont téléchargeables sur le site de l'éditeur. Un compte est nécessaire pour accéder aux téléchargements.

Le dernier firmware disponible pour ce modèle de PLC au moment de l'étude était celui déjà installé sur le PLC.

Le format du firmware est propre à l'éditeur. Il est composé principalement d'un en-tête comprenant une table de sections, suivi du contenu de chaque section. Un CRC-32 du contenu de chaque section est présent dans cet en-tête. Une section, nommée `A00000`, contient l'intégralité du code du firmware. La dernière section, `FW_SIG`, contient la signature.

5.1 Décompression du firmware

La section `A00000` est entièrement compressée. L'algorithme de compression est a priori inconnu, le code de décompression n'étant accessible qu'après décompression... Lors d'une mise à jour, le PLC vérifie la signature du firmware, décompresse le code de la section `A00000` et l'écrit sur une mémoire Flash.

La compression a donc dû être étudiée en boîte noire.

Certains blocs ont été particulièrement intéressants pour l'étude, car ils débutaient par des données dont la version décompressée était connue : il s'agissait principalement de pages du serveur Web (fichiers HTML ou CSS).

On remarque tout d'abord que certaines parties, notamment des morceaux de pages Web, apparaissent en clair. Toutefois, un caractère nul est inséré tous les 9 octets :

```
00 04 3C 68 74 6D 6C 3E 3C  ..<html><
00 62 6F 64 79 3E 0A 3C 74  .body>.<t
00 61 62 6C 65 20 63 65 6C  .able cel
00 6C 73 70 61 63 69 6E 67  .lspacing
00 3D 22 31 30 22 3E 0A 00  .="10">..
```

Les données semblent donc être stockées par blocs de 9 octets. Parfois, le texte est partiellement en clair. Ce n'est alors pas un caractère nul qui est inséré, mais par exemple 41 ou 10 aux lignes 2 et 4 de l'exemple suivant :

```
00 3C 6D 65 74 61 20 68 74  .<meta ht
41 74 08 22 63 6F 6E 74 03  At."cont.
00 2D 74 79 70 65 22 20 63  .-type" c
10 6F 6E 74 03 3D 22 74 65  .ont.="te
```

Le texte en clair est facile à deviner : il s'agit de la chaîne de caractères `<meta http-equiv="content-type" content="te`. Quelques octets ont été remplacés :

— `p-equiv=` a été remplacé par l'octet 0x08 ligne 2 ;

— `ent` a été remplacé par l'octet 0x03 aux lignes 2 et 4.

On remarque donc que ces octets ont été remplacés par leur longueur !

Les lignes où des octets ont été remplacés ne démarrent pas par un octet nul, mais par 0x41 et 0x10. Cet octet est en fait un masque : le ième bit de cet octet indique si le ième octet du bloc doit être recopié, ou s'il s'agit d'une longueur. Ainsi, un octet de masque à $0x41 = (1 << 6) + (1 << 0)$ signifie que les octets $2 = 8 - 6$ et $8 = 8 - 0$ contiennent des longueurs.

On peut donc déterminer la longueur de n'importe quel bloc compressé, et également décompresser une partie des données. Par exemple, on sait que l'exemple suivant :

```
40 73 09 68 61 6E 64 68 65  @s.handhe
00 6C 64 2C 20 6F 6E 6C 79  .ld, only
04 20 73 63 72 65 09 61 78  . scre.ax
```

sera décompressé pour obtenir un texte de la forme : `sXXXXXXXXXhandheld, only screXXXXXXXXXax`, où les octets à `X` sont inconnus.

Ceci fait très fortement penser à une compression de type LZ, où un dictionnaire est construit au fur et à mesure de la décompression.

Comment retrouver les octets manquants en fonction de leur longueur uniquement ? Les algorithmes LZ classiques encodent une longueur et une distance par rapport au bloc à recopier. Ici, seule une longueur est présente.

Une analyse manuelle assez fastidieuse a permis de deviner que les 4 octets précédant l'encodage d'une longueur étaient déjà présents dans les données précédemment décompressées. Ainsi, la précédente occurrence de "scre" était déjà suivie par une chaîne de longueur 9 correspondant au texte clair `media="handheld, only screen and (max-width: 767px)`.

Les données décompressées seront alors :

```
sXXXXXXXXXhandheld, only screen and (max
```

Ceci a permis de bien avancer. Toutefois, certaines données décompressées étaient incohérentes. Nous avons supposé que, pour des raisons de performance, l'algorithme de décompression ne faisait pas des recherches de 4 octets sur les données déjà décompressées pour obtenir les tokens à écrire, mais qu'une table de hachage était construite au fur et à mesure de la décompression. Cette hypothèse s'est avérée exacte. Le problème est que, pour des raisons d'occupation mémoire, la table de hachage est de petite taille. Des collisions se créent donc dans les entrées, ce qui fausse la décompression. Les erreurs s'enchaînent alors en cascade, et les données de sortie deviennent très vite complètement fausses.

La manière de construire la table de hachage étant inconnue, nous étions face à un mur. La solution est finalement apparue en étudiant les divers algorithmes LZ publics. La page de WikiBooks sur la compression [9] a été d'une grande utilité : elle mentionne l'algorithme LZP. C'est exactement cet algorithme, plus précisément LZP3, qui est utilisé dans le firmware.

La description de l'algorithme est détaillée dans [3]. Une implémentation a été développée, permettant l'accès en clair au firmware.

5.2 Signature du firmware

L'accès au code complet de ce firmware a été essentiel pour comprendre le mécanisme de signatures utilisé. Une analyse statique a montré qu'ECDSA-256 était utilisé. Afin d'éviter des problèmes de parsing, notamment sur la table d'en-têtes, une solution simple a été mise en œuvre : tout

le firmware est signé, excepté les 78 derniers octets, lesquels contiennent une signature de taille fixe. Les données présentes dans l'en-tête de la section « `FW_SIG` » ne sont donc en réalité pas prise en compte pour vérifier la signature.

La courbe et le générateur utilisés sont standard (ANSI X9.62 P-256). La fonction de hachage est SHA256.

Le mécanisme de vérification d'intégrité de chaque section, reposant sur CRC-32, n'est pas sûr d'un point de vue cryptographique, mais aucune exploitation n'est possible si le mécanisme de signature global du firmware n'a pas été contourné. La mise en œuvre de la vérification de signature a été analysée en détail. Aucune vulnérabilité n'a été trouvée sur ce mécanisme.

5.3 Organisation mémoire

Le fimware obtenu n'est pas dans un format connu (type ELF ou bFLT), et semble consister en un seul et unique exécutable. Pour explorer correctement ce firmware, l'idéal est d'avoir une idée du mapping mémoire du binaire.

En explorant le binaire, nous avons fini par retrouver une table des sections, dont le format est relativement simple. Elle peut être retrouvée en cherchant, par exemple, la chaîne de caractère `.text`, qui est présente dans la table des sections. En voici une traduction lisible :

Section name	Adress	Size	Permissions	Unknown
.exec_in_lomem	0x0	0x7f24	-- --x (0x1)	0x0
.bitable	0x40000	0x40	-- --x (0x1)	0x0
.sdramexec	0x40040	0x4d4	-- --x (0x1)	0x0
.syscall	0x40540	0x18	-- --x (0x1)	0x0
.th_initial	0x41040	0x2c70	i- --x (0x21)	0x0
.secinfo	0x43cc0	0x318	i- r-- (0x22)	0x0
.fixaddr	0x44000	0x0	-? --- (0x4)	0x0
.fixtype	0x44000	0x0	-? --- (0x4)	0x0
.text	0x44000	0xe490f0	i- --x (0x21)	0x0
.rodata	0xe8d100	0x3f4a6c	i- r-- (0x22)	0x0
.data	0x1281b80	0x27114	i- rw- (0x2a)	0x0
.bss	0x1e01040	0x7ce53c	-? -w- (0xc)	0x0
.uninitialized	0x3641040	0x394247c	-? -w- (0xc)	0x0
CLSI_CACHED_MEM_POOL	0x6f834c0	0x0	-? --- (0x4)	0x0
.dram_uncache	0x7ff0000	0x0	-? --- (0x4)	0x0
MAP_MAC_MEM	0x7ff0000	0x494	-? -w- (0xc)	0x0
.iram0	0x10030000	0x7aa0	-? -w- (0xc)	0x0
.iram1	0x10040000	0xc35c	-? -w- (0xc)	0x0
.crctable	0x1004f400	0x400	-? -w- (0xc)	0x0
.softboot	0x1004f800	0x700	-? -w- (0xc)	0x0
.bootinfo	0x1004ff00	0x1c	-? -w- (0xc)	0x0
.dtcm	0x10010000	0x2a00	-? -w- (0xc)	0x0

Le champ `Unknown` est toujours à zéro, mais est présent pour chaque entrée de la table. La permission `i` signifie « est initialisé quand chargé en mémoire », ce sont des données qui correspondent à des parties du binaire directement chargées en mémoire. Cette information est purement empirique, et il se trouve que la première section ne correspond pas à cette définition, puisqu'elle correspond à du code du firmware chargé en mémoire, mais n'a pas la permission `i`. La signification de la permission `?` est inconnue, il semblerait qu'elle autorise un certain type de lecture lorsque `i` n'est pas présent. Elle pourrait être interprétée comme « autorisation en lecture d'une donnée non initialisée ».

Voici comment le firmware est chargé en mémoire :

- Les 0x40 premiers octets sont un en-tête *a priori* non mappé.
- Les octets 0x40 à 0x7f64 (0x40 + la taille de `.exec_in_lomem`) sont chargés en 0.
- Les octets 0x7f64 à 0x8040 sont ignorés.
- Les autres sections sont chargées dans l'ordre, sans ignorer d'octet. Ainsi, les sections suivantes sont chargées à partir du firmware :
  - `.bitable` (un header)
  - `.sdramexec`
  - `.syscall`
  - `.th_initial` (du code initialisant, entre autres, la BSS)
  - `.secinfo` (description des sections)
  - `.text`
  - `.rodata`
  - `.data`

  Toutes les autres sont a priori non initialisées. La section `.bss` sera par exemple initialisée au boot par le code situé dans la section `.th_initial`.

Certaines de ces sections correspondent peut-être à des périphériques physiques. On notera d'ailleurs que le firmware déréférence régulièrement des adresses non *mappées* d'après la table des sections, faisant penser à un accès à des composants hardware, par exemple des accès proches de l'adresse `0xffffffff`. Le début des données est un en-tête décrivant l'organisation de la mémoire du PLC. Il est suivi par le code. On peut donc maintenant procéder à une analyse statique du code complet du PLC.

Malheureusement, les algorithmes obfusqués dans le logiciel de supervision le sont également sur le PLC.

6 Conclusion

Dans cet article, nous avons exposé notre démarche dans la rétro conception d'un système industriel. La première phase de fuzzing nous a permis de nous familiariser avec l'architecture et a permis d'identifier des points d'intérêt dans les technologies manipulées.

La rétro conception en boîte noire nous a rapidement apporté tout ce qu'il nous fallait pour faire une analyse en boîte blanche ciblée de la sécurité du protocole. Cette dernière phase a permis de découvrir une bonne partie du cryptosystème mis en place, de découvrir une vulnérabilité et de monter une attaque opérationnelle l'exploitant. L'analyse du firmware nous a ouvert de nouveaux horizons en ce qui concerne la recherche de vulnérabilités sur le PLC.

Ces informations nous permettront par la suite d'aller plus loin dans la compréhension de ces technologies, permettant par exemple de faire du fuzzing plus ciblé en connaissant tous les formats de données utilisés par le protocole, ou d'essayer de compiler des programmes utilisateurs arbitraires.

Contrairement à ce qui a pu se voir précédemment, ce constructeur a fait un effort très conséquent en ce qui concerne la sécurité de ses équipements, et même s'il reste du chemin avant d'arriver à un niveau de confiance proche de ce que l'on peut retrouver dans des systèmes d'information classiques, les choses évoluent dans le bon sens.

Remerciements

Les auteurs tiennent à remercier Pierre Capillon et Florent Marceau pour leurs précieux conseils lors de la rédaction de ce document.

Références

1. Aki Helin. Radamsa. `https://code.google.com/p/ouspg/wiki/Radamsa`.
2. Aleksandr Timorin. SCADA deep inside : protocols and security mechanisms. `http://fr.slideshare.net/AlexanderTimorin/scada-deep-inside-protocols-and-security-mechanisms-40672525`, 2014.
3. Charles Bloom. Lzp : A new data compression algorithm. In James A. Storer and Martin Cohn, editors, *Data Compression Conference*, page 425. IEEE Computer Society, 1996.
4. Florent Monjalet. Hexlighter. `https://github.com/fmonjalet/hexlighter`.
5. Luigi Auriemma. signsrch. `http://aluigi.altervista.org/mytoolz/signsrch.zip`.

6. Orion Ragozin. Introduction à la cybersécurité des systèmes d'information industriels : Méthodes de classification et mesures principales. *Misc*, (74) :27–33, July 2014.
7. Rob Savoye. Reverse Engineering of Proprietary Protocols, Tools and Techniques. In *FOSDEM*, 2009.
8. Symantec. W32Stuxnet. `http://www.symantec.com/security_response/writeup.jsp?docid=2010-071400-3123-99`, 2010.
9. WikiBooks. Data compression/dictionary compression. `http://en.wikibooks.org/wiki/Data_Compression/Dictionary_compression`, 2014.