

Wirego - Un outil de développement de plugins Wireshark

Benoît Girard
bgirard@quarkslab.com

Quarkslab

Résumé. Wirego est un outil permettant de développer rapidement des plugins Wireshark [3] dans différents langages de programmation. Là où traditionnellement Wireshark nécessite le développement de plugins en langage C via une API puissante mais complexe, ou en LUA, Wirego permet rapidement de mettre en place l'analyse d'un protocole, notamment dans le cadre d'une rétro-ingénierie. Wirego s'appuie sur un plugin Wireshark traditionnel, un package (Go, Python, Rust...) réalisant l'interface et le code développé par l'utilisateur final.

1 Introduction

1.1 État de l'art

Lorsqu'il est nécessaire d'effectuer la rétro-ingénierie d'un protocole réseau, le point de départ est bien souvent une capture de type pcap. L'outil de référence pour visualiser celle-ci est Wireshark [3]. L'outil prend en charge la plupart des protocoles usuels et permet d'accéder facilement à la liste des paquets, les informations réseau et les payloads. Parallèlement, on développe souvent un "parseur" dans un langage facile



Fig. 1. Wireshark

d'accès : Python ou Go, au fur et à mesure que notre compréhension du protocole avance.

Lorsque le protocole est particulièrement complexe, il devient difficile de déterminer quels paquets ont été effectivement traités et ceux qui comportent encore des zones d'incertitude. A ce stade, il semblerait logique de développer un plugin dédié pour Wireshark, afin de visualiser

directement les résultats dans l'outil.

Wireshark a été publié initialement en 1998 avec dès le départ la volonté de permettre d'intégrer "facilement" des analyseurs protocolaires tiers. Bien que l'effort soit louable, après plus de 25 ans d'existence, il apparaît que peu de personnes se sont lancés dans l'aventure.

Le développement d'un nouveau plugin requiert le téléchargement du code source de Wireshark, sa compilation, puis le développement du plugin en tant que tel, en s'appuyant sur une documentation fréquemment incomplète et des API complexes. Dans le cadre de la rétro-ingénierie d'un protocole le plugin sera souvent "jetable", il ne s'agit ici que d'un outil d'aide à la compréhension. Il est alors contre productif de mettre en pause les travaux pour se plonger dans l'analyse et la compréhension des mécanismes internes de Wireshark.

Conscients du problème, les développeurs de Wireshark ont mis en place la possibilité de développer des plugins en langage LUA dès la version 0.99.4 sortie en 2006. Là encore, le langage étant peu utilisé, il n'est pas toujours pertinent de se lancer dans l'apprentissage d'un nouveau langage en plein milieu de travaux de rétro-ingénierie.

Parallèlement un support pour Python avait été mis en place, avant d'être finalement abandonné quelques années plus tard faute de mainteneur (voir PyReshark [2]).

Que l'on fasse le choix du développement d'un plugin en langage C ou d'utiliser l'interface LUA, un autre problème apparaît rapidement : le manque d'outils et de fonctions tierces. Si une bibliothèque tierce est nécessaire pour traiter un paquet (chiffrement, compression, encodage ou autre) il sera alors nécessaire de chercher le bon outil, qui ne sera pas toujours simple à utiliser, ni même disponible (notamment dans le cas de LUA).

Ces différents problèmes expliquent pourquoi le développement de plugins Wireshark reste relativement rare dans le cadre d'analyse de protocoles, malgré l'utilisation massive de l'outil natif au sein de la communauté.

2 Problématique et Architecture

Une première version de Wirego publiée fin 2023 permettait de développer des plugins Wireshark en Go. Un plugin en C, chargé par Wireshark ouvrait dynamiquement une librairie dynamique qui embarquait le code utilisateur via des bindings en C. Cette version était parfaitement fonctionnelle mais avait une limitation importante :

l'utilisateur final devait impérativement maîtriser le langage Go. En remplaçant une dépendance depuis le langage LUA vers le Go, on ne se contentait que de décaler le problème.

Une évolution de cette version permettant l'ouverture dynamique de plugins écrits dans d'autres langages (Rust notamment) a été étudiée, mais chaque intégration d'un nouveau langage ajoutait une nouvelle couche de complexité rendant l'ensemble extrêmement opaque.

La version 2.0 de Wirego permet de traiter cette problématique en s'appuyant sur une couche intermédiaire : Zero-MQ.

2.1 Zero-MQ

Zero-MQ, noté également 0MQ ou ZMQ, (voir [4]) est un protocole de communication asynchrone orienté sur la performance. Des primitives permettent facilement de mettre en place des schémas de communication classiques tels que :

- PUB/SUB : Un "publisher" (PUB) envoie des messages sur un canal, les "subscribers" (SUB) les reçoivent
- REQ/REP : Un processus envoie une requête (REQ), un autre y répond (REP)
- Pipelines : Qui permet de précâbler un canal de transmission



Fig. 2. ZMQ

ZMQ met en oeuvre un système de "brokers", permettant de gérer des files d'attente, des priorités, des mécanismes de rejeu et d'une manière générale des concepts de QOS (Quality of Service).

La librairie est activement maintenue et est disponible dans un très grand nombre de langages de programmation : C, C++, CL, Delphi, Erlang, Elixir, Felix, Go, Haskell, Haxe, Java, Julia, Lua, Node.js, Objective-C, Perl, PHP, Python, Q, Racket, Ruby, Rust, Scala, Tcl, OCaml, Ada, Basic, C#, F#, ooc.

Il est à noter également que la documentation de ZMQ est absolument

remarquable, très détaillée, très claire et parfois même drôle. Toutes les personnes ayant un jour eu à faire avec une RFC apprécieront.

2.2 Architecture

Wirego s'appuie sur une architecture à trois étages :

- Un plugin Wireshark écrit en C, chargé de transmettre les requêtes de Wireshark vers un endpoint ZMQ
- Un "package" recevant ces requêtes ZMQ et effectuant des appels vers le code utilisateur, dans un langage donné
- Le code développé par l'utilisateur, dans un langage donné

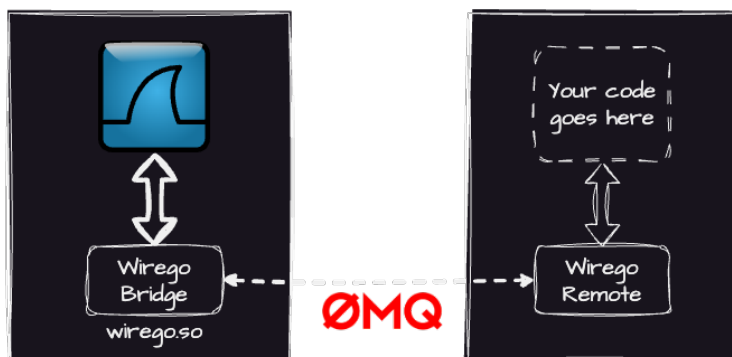


Fig. 3. Architecture

Dans sa version 2.0 Wirego fournit, en plus du plugin Wireshark, un package en Go ainsi que des exemples de code utilisateur.

On utilisera dans la suite du document la terminologie suivante :

- Wirego bridge : le plugin écrit en langage C, chargé par Wireshark
- Wirego remote : le plugin écrit par l'utilisateur, s'appuyant un package disponible pour un langage donné
- Package Wirego : un package pour un langage donné facilitant l'utilisation

2.3 Fonctionnement des plugins Wireshark

Un plugin Wireshark doit impérativement implémenter trois fonctions qui seront appelées successivement par l'outil.

Listing 1: API Wireshark

```
1 void proto_register_xxxx(void);
2 void proto_reg_handoff_xxxx(void);
3 int dissect_xxxx(tvbuff_t *tvb, packet_info *pinfo, proto_tree
  ↳ *tree _U_, void *data _U_);
```

La fonction **proto_register_xxxx** est appelée au chargement du plugin par Wireshark.

Le plugin doit effectuer les tâches suivantes :

- Enregistrer le plugin via un appel à **proto_register_protocol**
- Déclarer les champs personnalisés qui pourront être retournés

Lors de l'enregistrement du plugin on fournit le nom "affichable" du plugin, un nom interne et un filtre. Le filtre permettra ultérieurement de filtrer le trafic dans Wireshark. La déclaration des champs personnalisés susceptibles d'être utilisés lors de l'analyse d'un paquet permet notamment à Wireshark de préparer l'interface graphique.

Une fois que tous les plugins sont initialisés, Wireshark fait appel aux fonctions **proto_reg_handoff_xxxx**. Cet appel permet à un plugin :

- D'enregistrer la fonction de dissection via **create_dissector_handle**
- De déclarer des filtres de détection, qui permettront de router le trafic vers cette fonction : **dissector_add_uint** et **dissector_add_string**

Un plugin Wireshark se focalise sur un protocole donné. La définition d'un filtre via **dissector_add_uint** ou **dissector_add_string** permet par exemple de s'intéresser au trafic sur un port (**tcp.port == 25**) ou sur une IP (**ip.addr == 192.168.1.255**) donnée. Dans certains cas un simple filtre Wireshark n'est pas suffisant pour caractériser un protocole. Il est alors nécessaire d'implémenter une fonction d'heuristique permettant d'effectuer une pré-analyse d'un paquet afin de déterminer si celui-ci correspond au protocole traité par le plugin ou non.

Afin de déclarer auprès de Wireshark une fonction de détection par heuristique, il est nécessaire de faire un appel à **heur_dissector_add** en fournissant en plus de la callback de détection une liste de filtres définissant les protocoles parents susceptibles d'embarquer notre protocole.

Wireshark n'est pas un système de type DPI (Deep Packet Inspection), la stratégie d'analyse est par conséquent assez limitée. Il n'y a pas de gestion de priorités entre protocoles, il sera donc par exemple impossible de se substituer au module HTTP sur le port 80. Tous les

filtres classiques (sans heuristique) sont évalués avant les fonctions de détection par heuristique. Tous les plugins internes à Wireshark sont évalués avant les plugins externes.

Wireshark ne met pas en oeuvre de mécanisme de suivi de session. La fonction de détection par heuristique sera donc appelée sur chaque paquet de la session (TCP notamment).

Au final il sera parfois nécessaire de désactiver manuellement certains modules de Wireshark afin de pouvoir enfin traiter le protocole recherché.

Pour finir, lors de l'affichage d'un pcap ou lors d'une capture "live", la fonction **dissect_xxxx** est appelée si le paquet satisfait les contraintes des filtres définies par le plugin. La fonction **dissector** devra alors utiliser un certain nombre de fonctions pour ajouter un "noeud" d'affichage puis lister les champs personnalisés qui ont été identifiés.

Pour des raisons de facilité de lecture nous n'entrerons pas ici dans les détails techniques des différentes (et nombreuses) API proposées. Le lecteur est invité à se reporter à la documentation dédiée de Wireshark sur le sujet [1] et plus largement à se référer au code source des modules existant pour les fonctions avancées.

2.4 Fonctionnement du plugin Wirego

Le plugin Wirego est développé en C et implémente les API requises par Wireshark. Sa tâche principale est de rediriger les appels effectués vers un endpoint ZMQ.

Initialisation `proto_register_wirego` Lors de l'initialisation, la fonction **`proto_register_wirego`** est appelée.

Wirego bridge tente d'ouvrir une connexion vers le endpoint ZMQ définit dans les propriétés du plugin. En cas d'échec (le "Wirego remote" n'est pas lancé), le plugin est alors désactivé.

La fonction **`proto_register_protocol`** est ensuite appelée en fournissant les informations collectées auprès du Wirego remote (nom du plugin, filtre...). On demande à cette étape au plugin utilisateur de fournir la liste des champs personnalisés susceptibles d'être définis lors de la dissection d'un paquet. Ceux-ci sont déclarés auprès de Wireshark via des API dédiées.

Enregistrement `proto_reg_handoff_wirego` La seconde passe d'initialisation nous permet d'enregistrer un dissector et de définir les critères

qui permettront à Wireshark de déterminer si un paquet doit nous être envoyé.

Les filtres sont récupérés auprès du plugin utilisateur puis transmis à Wireshark. La fonction de dissection enregistrée est définie dans le plugin Wirego, elle se chargera de faire une partie des conversions nécessaires pour faciliter les échanges via le tunnel ZMQ.

Dissection dissect_wirego Lorsque Wireshark identifie un paquet satisfaisant les filtres définis, un appel à la callback de dissection est effectué. On ne souhaite pas transférer ici cet appel tel quel à l'utilisateur : les arguments fournis à la fonction s'appuient sur des types de données complexes avec des API dédiées permettant de les traiter. Le plugin Wirego va pré-traiter ces arguments pour fournir au plugin utilisateur des arguments simples, suffisants dans la plupart des cas d'utilisation.

Un choix stratégique est ici nécessaire car en effet, le fait de simplifier les arguments passés à la fonction de dissection limite les cas d'usage pouvant être pris en charge par le plugin utilisateur. L'API de Wireshark native est puissante et donc complexe. L'objectif de Wirego est de permettre de développer rapidement un plugin dans le cadre d'un reverse de protocole. Mettre en place des "bindings" complets aurait un effet contre productif en rendant le "coût d'entrée" prohibitif pour une personne ne développant que ponctuellement des plugins. Au final le système mis en place permet de prendre en charge la plupart des cas "simples".

Un certain nombre d'API Wireshark mises à disposition des plugins utilisent des types complexes, avec des structures et des tableaux imbriqués. Dans notre cas d'usage, nous devons transmettre ces éléments via un canal ZMQ. En bout de chaîne, un package écrit dans un langage quelconque doit alors convertir ces commandes ZMQ en un tout intelligible et cohérent, mis à disposition de l'utilisateur final.

Il aurait été possible d'imposer une technologie de sérialisation mais cela aurait ajouté un niveau de complexité supplémentaire pour le mainteneur de package dans le langage de programmation cible. Par ailleurs ZMQ est disponible dans de nombreux langages, alors que les bibliothèques de sérialisation couvrent généralement un nombre plus réduit de langages.

Le choix a été fait de :

- S'appuyer sur les "frames" propres à ZMQ pour transmettre les valeurs des structures une à une
- Dérouler les tableaux sous la forme d'une succession de requêtes permettant d'accéder aux éléments

Ce choix a un impact sur les performances et multiplie le nombre de requêtes ZMQ effectuées. En contrepartie, il simplifie grandement l'implémentation d'un nouveau package pour un langage de programmation non encore supporté.

Paramètres Au delà des aspects purement protocolaires, un plugin Wireshark a la possibilité de définir une page de "paramètres" permettant de configurer certaines options de celui-ci. Dans notre cas le plugin Wirego définit une page permettant de définir le endpoint ZMQ partagé entre Wirego bridge et Wirego remote (voir 4). Pour des raisons techniques liées à l'architecture de Wireshark, il sera nécessaire de redémarrer Wireshark en cas de modification des préférences.

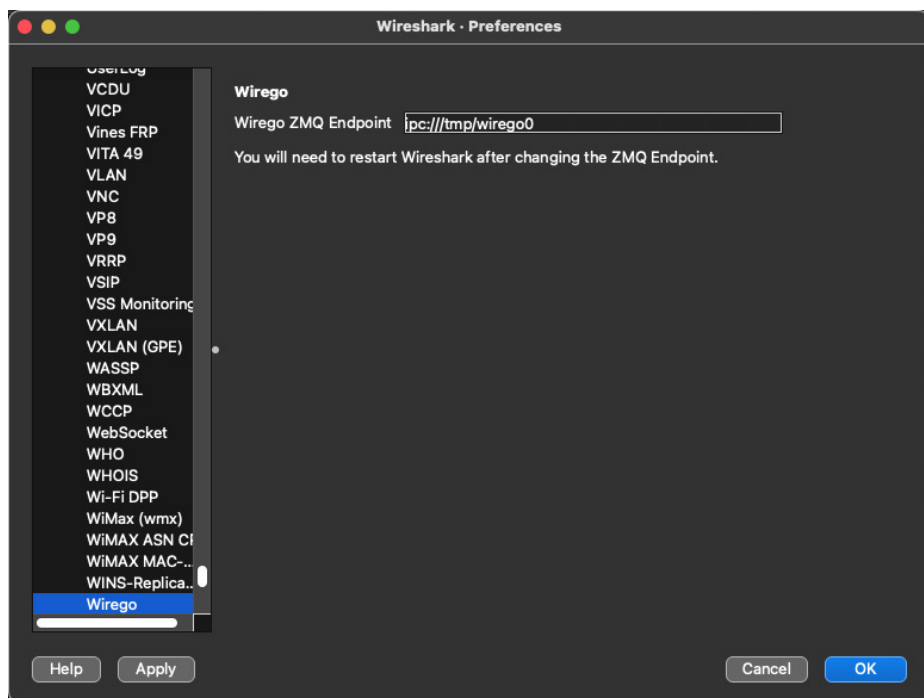


Fig. 4. Configuration du plugin Wirego

2.5 Fonctionnement du package Wirego pour Go

Le plugin Wirego pour Wireshark (Wirego bridge) transmet les appels de Wireshark vers un endpoint ZMQ via un schéma REQ/REP.

Un package Go disponible permet de recevoir ces différents appels et les convertir vers une API simple s'appuyant sur l'implémentation d'une interface par l'utilisateur final. Nous venons de voir qu'afin de ne pas utiliser de librairie de serialisation, Wirego expose les structures en "Frames ZMQ" et déroule les tableaux en appels successifs. Le package Wirego se charge de faire l'opération inverse : il collecte auprès de l'utilisateur des types de données complexes (listes de structures), les stocke localement et y accède au fil des appels ZMQ.

Afin de gérer au mieux un maximum de protocoles, Wireshark effectue une analyse des paquets en deux passes. Lors de l'ouverture d'un pcap notamment, tous les paquets sont passés aux dissectors une première fois. Lorsque l'utilisateur fait défiler la liste des paquets, ou lorsqu'il en sélectionne un, les fonctions dissector sont à nouveau appelées pour les paquets visibles. Ce mécanisme peut engendrer un nombre significatif d'appels aux fonctions dissector. Il est recommandé, lorsque cela est pertinent, d'implémenter un cache afin de ne pas réanalyser un paquet qui aurait déjà été traité. Wirego laisse ici le choix à l'utilisateur, un cache est présent par défaut mais désactivable si besoin : **wirego.ResultatsCacheEnable(false)**.

3 Implémentation d'un plugin utilisateur en Go

3.1 Enregistrement

D'une manière générale, un package Go fournit à un programme des fonctions additionnelles. Dans notre cas c'est l'inverse : le package wirego fournit l'essentiel des fonctionnalités, le code utilisateur se charge d'implémenter une sous partie des fonctionnalités via une interface. Chaque plugin utilisateur devra mettre en place, en plus de l'interface définie au paragraphe suivant (voir 3.2), quelques mécanismes permettant d'enregistrer le code utilisateur auprès du package wirego (voir 2).

Listing 2: Implementation d'un plugin utilisateur

```
1  package main
2
3  import (
4      "fmt"
5
6      "github.com/quarkslab/wirego/wirego/wirego"
7  )
8
9  // Define here enum identifiers, used to refer to a specific
↪  field
10 const (
11     FieldIdCustom1          wirego.FieldId = 1
12     FieldIdCustom2          wirego.FieldId = 2
13     FieldIdCustomWithSubFields wirego.FieldId = 3
14 )
15
16 // Since we implement the wirego.WiregoInterface we need some
↪  structure to hold it.
17 type WiregoMinimalExample struct {
18 }
19
20 func main() {
21     var wge WiregoMinimalExample
22
23     wg, err := wirego.New("ipc:///tmp/wirego0", false, wge)
24     if err != nil {
25         fmt.Println(err)
26         return
27     }
28     wg.ResultsCacheEnable(false)
29
30     wg.Listen()
31 }[...]
```

On définit ici une structure (nommée **WiregoMinimalExample** pour l'exemple) qui nous permettra par la suite de porter l'interface requise. Notre fonction **main()** crée une instance du package wirego et fournit en argument le endpoint ZMQ sur lequel il faudra écouter. On précise également l'objet implémentant l'interface et qui recevra les appels provenant de Wirego bridge.

On fait ici le choix de désactiver le cache des résultats de dissection : un même paquet pourra donc potentiellement être analysé plusieurs fois.

3.2 L'interface WiregoInterface

L'utilisateur devra implémenter l'interface suivante :

Listing 3: Interface Wirego

```
1  type WiregoInterface interface {
2      GetName() string
3      GetFilter() string
4      GetFields() []WiresharkField
5      GetDetectionFilters() []DetectionFilter
6      GetDetectionHeuristicsParents() []string
7      DetectionHeuristic(packetNumber int, src string, dst string,
↪  stack string, packet []byte) bool
8      DissectPacket(packetNumber int, src string, dst string, stack
↪  string, packet []byte) *DissectResult
9  }
```

Le nombre de fonctions à implémenter est réduit au minimum et un exemple de plugin utilisateur comportant moins de 100 lignes de code est fourni [5].

La fonction **GetName()** permet de définir le nom du plugin tel qu'il apparaîtra dans Wireshark (Exemple : "Linksys Update Protocol").

GetFilter() définit le nom du filtre qui permettra à l'utilisateur de filtrer le trafic dans Wireshark, via la barre de filtres (Exemple : "linksysupd"). Le filtre défini permettra également de créer des filtres plus complexes, s'appuyant sur les données extraites par le dissector et définies par le plugin (Exemple : "linksysupd.seq_num == 12").

GetFields() permet à l'utilisateur de définir les types de données (fields) qu'il sera susceptible de retourner lors d'une dissection. Cette liste doit impérativement être exhaustive, il ne sera pas possible de définir de nouveaux types dynamiquement à la suite de la dissection d'un paquet. On retourne ici une liste de **WiresharkField** définis comme suit :

Listing 4: Définition d'un champ utilisateur

```
1  type FieldId int
2  type WiresharkField struct {
3      WiregoFieldId FieldId
4      Name           string
5      Filter          string
6      ValueType       ValueType
7      DisplayMode     DisplayMode
8  }
```

Le champ **FieldId** doit être considéré comme un "enum" permettant facilement d'identifier un champ. L'utilisateur est libre de définir autant de **FieldId** qu'il le souhaite. On définit ensuite un nom "**Name**" (Exemple : "Sequence Number") pour le champ en question et un nom de filtre "**Filter**" (Exemple : "seq_num"). Un champ de résultat peut faire référence à une partie du payload. Il est donc nécessaire d'indiquer à Wireshark le type de donnée (**ValueType**) et le mode d'affichage (**DisplayMode**).

La fonction **GetDissectorFilter** permet de définir les filtres qui sélectionneront les paquets à envoyer vers le dissector. On retourne ici une liste de **DissectorFilter** définis de la façon suivante :

Listing 5: Définition d'un filtre de dissection

```
1  type DissectorFilter struct {
2      FilterType DissectorFilterType
3      Name       string
4      ValueInt   int
5      ValueString string
6  }
```

Deux types de filtres peuvent être définis, le champ **FilterType** permet de choisir entre un type entier (Exemple : "udp.port == 17") ou un type chaîne (Exemple : "bluetooth.uuid = '1234'"). On précise ensuite le nom du filtre de dissection à appliquer via le champ **Name** (Exemple : "tcp.port") puis la valeur en question, en fonction du type choisi.

Comme nous l'avons vu précédemment, l'utilisation de simple filtres pour détecter un protocole peut s'avérer insuffisante. Pour prendre en charge les heuristiques, il sera nécessaire d'implémenter la fonction **GetDetectionHeuristicsParents** qui retournera la liste des parents susceptibles d'embarquer notre protocole.

Si un paquet correspond à un des protocoles parents défini, on recevra

alors le paquet pour détection via **DetectionHeuristic**. Les arguments de cette fonction sont identiques à ceux de la fonction **DissectPacket** décrite ci-après. En retour de la fonction de détection par heuristique on renvoie un simple booléen permettant de préciser si le paquet correspond à notre protocole ou non.

L'essentiel du code effectif d'un plugin utilisateur se situe dans la fonction **DissectPacket**. L'utilisateur reçoit en argument de cette fonction :

- **packetNumber** : le numéro du paquet dans le pcap ou la capture en cours
- **src** : l'adresse réseau source (qui peut être en fonction des couches réseau une adresse IP ou MAC)
- **dst** : l'adresse réseau de destination (qui peut être en fonction des couches réseau une adresse IP ou MAC)
- **stack** : l'empilement des couches protocolaires
- **packet** : le payload du paquet à analyser

La "stack" réseau fournie utilise la nomenclature propre à Wireshark, elle démarrera donc avec "frame" suivi d'un mot clef pour chaque protocole identifié (Exemple : "frame.eth.ethertype.ip.udp.linksysupd")

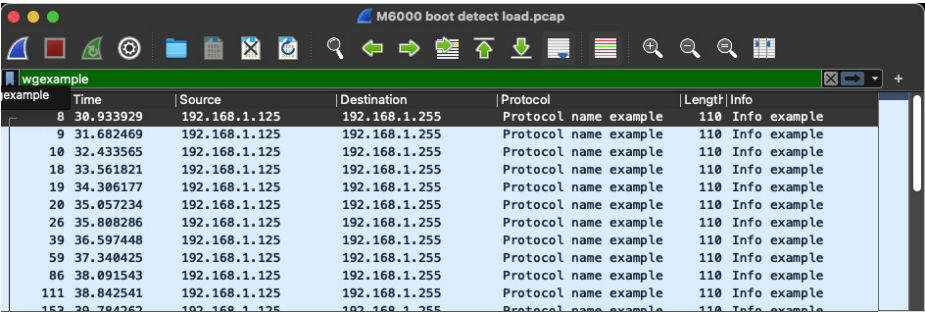
L'utilisateur analyse le paquet fourni puis retourne un **DissectResult** :

Listing 6: Définition d'un résultat de dissection

```
1 type DissectResult struct {
2     Protocol string
3     Info      string
4     Fields    []DissectField
5 }
6 type DissectField struct {
7     WiregoFieldId FieldId
8     Offset         int
9     Length         int
10    SubFields      []DissectField
11 }
```

Un résultat de dissection permet de définir deux champs génériques d'identification **Protocol** et **Info** qui seront affichés directement dans les colonnes de description du paquet (voir 5).

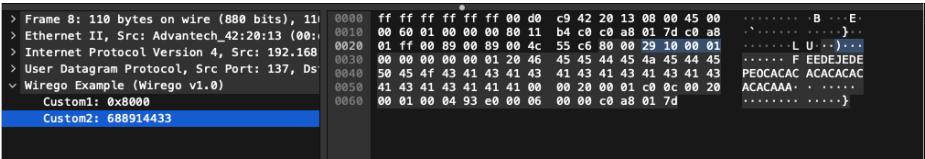
Le résultat peut également fournir une liste de **DissectField** qui feront référence au type de champ personnalisé défini précédemment (voir 3.2) via l'enum **WiregoFieldId**. On précise ici l'offset du champ dans le payload et la taille de celui-ci afin de permettre à Wireshark d'afficher



example	Time	Source	Destination	Protocol	Length	Info
8	30.933929	192.168.1.125	192.168.1.255	Protocol name example	110	Info example
9	31.682469	192.168.1.125	192.168.1.255	Protocol name example	110	Info example
10	32.433565	192.168.1.125	192.168.1.255	Protocol name example	110	Info example
18	33.561821	192.168.1.125	192.168.1.255	Protocol name example	110	Info example
19	34.306177	192.168.1.125	192.168.1.255	Protocol name example	110	Info example
20	35.057234	192.168.1.125	192.168.1.255	Protocol name example	110	Info example
26	35.808286	192.168.1.125	192.168.1.255	Protocol name example	110	Info example
39	36.597448	192.168.1.125	192.168.1.255	Protocol name example	110	Info example
59	37.340425	192.168.1.125	192.168.1.255	Protocol name example	110	Info example
86	38.091543	192.168.1.125	192.168.1.255	Protocol name example	110	Info example
111	38.842541	192.168.1.125	192.168.1.255	Protocol name example	110	Info example
152	39.784262	192.168.1.125	192.168.1.255	Protocol name example	110	Info example

Fig. 5. Champs Protocol et Info

une description détaillée du paquet analysé et faire référence aux données sous-jacentes.



Frame 8: 110 bytes on wire (880 bits), 110 bytes captured (880 bits) on interface 0	Time	Source	Destination	Protocol	Length	Info
> Ethernet II, Src: Advantech_42:20:13 (00:0c:00:00:00:00), Dst: 01:00:5e:00:00:00	30.933929	192.168.1.125	192.168.1.255	Ethernet II	14	Src: Advantech_42:20:13 (00:0c:00:00:00:00), Dst: 01:00:5e:00:00:00
> Internet Protocol Version 4, Src: 192.168.1.125, Dst: 192.168.1.255	30.933929	192.168.1.125	192.168.1.255	Internet Protocol Version 4	20	Src: 192.168.1.125, Dst: 192.168.1.255
> User Datagram Protocol, Src Port: 137, Dst Port: 137	30.933929	192.168.1.125	192.168.1.255	User Datagram Protocol	28	Src Port: 137, Dst Port: 137
> Wirego Example (Wirego v1.0)	30.933929	192.168.1.125	192.168.1.255	Wirego Example (Wirego v1.0)	110	Custom1: 0x8000, Custom2: 688914433

Fig. 6. Affichage d'un champ personnalisé

Chaque champ définit peut éventuellement embarquer des sous-champs via l'utilisation de **SubFields** []**DissectField**. Les sous-champs seront alors affichés dans Wireshark sous la forme d'une arborescence.

3.3 Limitations

Dans sa version 2.0, Wirego possède un certain nombre de limitations connues, et parfois même assumées.

- Le choix de ne pas utiliser de mécanisme de serialisation pour les types complexes génère un nombre de requêtes ZMQ parfois important. Il s'agit ici d'un choix assumé permettant de favoriser la création de "packages" dans des langages divers.
- Utiliser un protocole de communication plutôt qu'un "binding" direct comme c'était le cas avec Wirego 1.0 limite les performances. Il s'agit ici d'un choix assumé, la mise en palce de "bindings" n'étant pas réalisable dans tous les langages.

```
> Frame 1: 92 bytes on wire (736 bits), 92 bytes captured (736 bits) on interface 0
> Ethernet II, Src: Advantech_42:20:13 (00:d0:c9:42:20:13), Dst: 192.168.1.125
> Internet Protocol Version 4, Src: 192.168.1.125, Dst: 192.168.1.125
> User Datagram Protocol, Src Port: 137, Dst Port: 137
  ▾ Wirego Minimal Example (Wirego v2.0)
    Custom1: 0x8214
    Custom2: 17825793
    ▾ CustomWith Subs: 0x00000000
      Custom1: 0x00
      Custom1: 0x00
```

Fig. 7. Arborescence

- Wirego 2.0 ne supporte pas les paquets répartis sur plusieurs paquets issues de la couche parente. C'est notamment le cas lors de la présence de couches de compression ou de chiffrement.
- Si Wireshark est démarré avant que le "Wirego remote" ne soit lancé, alors le plugin Wirego sera désactivé. Il s'agit ici d'une contrainte technique propre à Wireshark : lors de son démarrage Wireshark a besoin de collecter diverses informations auprès des plugins (nom du plugin, champs personnalisés...).
- Si l'utilisateur définit un endpoint de type **tcp** `://127.0.0.1 :5555` et décide de lancer une capture sur l'interface "localhost", il prend alors le risque d'ouvrir un trou noir.
- Lors de la rédaction de cet article, seul le package pour **Go** est disponible. Le support des langages Python et Rust est envisagé à court terme.

3.4 Conclusion

Wirego est un outil permettant de développer rapidement un plugin Wireshark, lors d'une opération de rétro-ingénierie protocolaire notamment. Le plugin wirego [6], développé en C ne nécessite d'être compilé qu'une seule fois et des versions précompilées sont disponibles pour la plupart des systèmes d'exploitation. L'utilisateur final n'a besoin que de s'interfacer avec un package dans un langage qu'il maîtrise, sans se préoccuper des mécanismes internes de Wireshark ni des ajustements nécessaires au sein de Wirego.

Références

1. Documentation du développeur. https://www.wireshark.org/docs/wsdg_html_chunked/ChDissectAdd.html.
2. PyReshark. <https://github.com/ashdnazg/pyreshark/>.
3. Site web de Wireshark. <https://www.wireshark.org/>.
4. ZMQ. <https://zeromq.org/get-started/>.
5. Neb. Exemples de plugin Wirego. <https://github.com/quarkslab/wirego/tree/main/examples>.
6. Neb. Wirego v2.0. <https://github.com/quarkslab/wirego/>.