Simulateur de Botnet

1 Compilation

Deux versions du simulateur ont été programmés en C++, une qui utilise un thread pour chaque bot du botnet et l'autre sans. Dans le premier cas cela permet le calcul en parallèle, rendant la simulation plus proche de la réalité, chaque bot fonctionnant de manière indépendante, dans le deuxième cas, les actions des bots sont effectués séquentiellement, le ou les actions du bot numéroté 3 sont effectués après le ou les actions du bot 2, ce qui est moins réaliste.

Cette deuxième version est utile lorsque l'on souhaite lancer la simulation sur un seul processeur, puisqu'elle nécessite moins de ressources (on économise le changement de contexte effectué par l'OS lorsqu'il bascule d'un thread à l'autre). Dans le CMakeFile.txt on ajoutera ou pas la macro THREAD_VERSION dans les options de gcc en fonction de la version du simulateur que l'on souhaite compiler.

```
set (CMAKE_CXX_FLAGS "-Wall -W -Wno-deprecated -D THREAD_VERSION")
```

La compilation du programme nécessite les librairies Boost, QT et VTK (avec support Qt).

2 Classe Bot

Tout les bots du Botnet (spammers, repeaters, attackers, protecters, c&c dans le cas de Waledac) dérivent de cette classe.

```
class Bot : public
boost::enable_shared_from_this< Bot >
{

private :
std::string m_id;
bot_status m_status;

public :
Bot();
const std::string& id() const;

virtual bot_status& status();
bool is_compromised();

virtual response_code
send_message(message_code message);

virtual void init(bot_t& server,
bots_t& plist, bots_t& rlist) = 0;

#ifdef THREAD_VERSION
virtual void execute() = 0;
virtual void start() = 0;
virtual void wait() = 0;
#endif

};
```

Chaque bot à ainsi un identifiant (m_id) et un status (m_status) accessiblement respectivement par le biais des fonctions id() et status(). La fonction is_compromised() permet de savoir si un bot est entre les mains de l'attaquant. Le bot est dit compromit par l'attaquant si et seulement si le statut du bot vaut COMPROMISED et a ce moment le fonctionnement du Bot est stoppé (il ne peut pas repasser à l'ennemi).

Chaque Bot peut effectuer des "actions", pour l'instant seul l'envoi d'un message est possible (par le biais de la fonction send_message()).

3 Classe Spammer

Cette classe implémente les fonctionnalitées d'un Bot de type spammer.

```
class Spammer : public Bot
{
private:
bots_t m_rlist;
void update_status(response_code code);
#ifdef THREAD_VERSION
boost::shared_ptr< boost::thread >
m_spammer_thread;
#endif

public:
Spammer();
void update_rlist();

bots_t rlist();

virtual response_code
send_message(message_code message);

virtual void init(bot_t& server,
bots_t& plist, bots_t& rlist);

#ifdef THREAD_VERSION
virtual void execute();
virtual void start();
virtual void wait();
#endif

};
```

Chaque spammer à une rlist (m_rlist) accessible par le biais de la fonction rlist() et qui est mise à jour grace à la fonction update_rlist(). Dans un botnet de type waledac c'est le spammer qui se connecte au répéteur, dans notre cas cette action est effectué en appellant la fonction send_message(MESSAGE_TASKREQ) qui signifie qu'il demande une tache.

4 Classe Repeater

Cette classe implémente les fonctionnalitées d'un Bot de type Repeater.

```
class Repeater : public Bot
{
private:
bots_t m_rlist;
bots_t m_plist;
#ifdef THREAD_VERSION
boost::shared_ptr<boost::thread>
m_repeater_thread;
#endif

public:
Repeater();
virtual void update_rlist();
virtual void update_plist();

virtual bots_t sub_rlist();
virtual bots_t sub_plist();

bots_t plist();
bots_t rlist();

virtual response_code
send_message(message_code message);

virtual void init(bot_t& server,
bots_t& plist, bots_t& rlist);

#ifdef THREAD_VERSION
virtual void execute();
virtual void start();
virtual void wait();
#endif
};
```

Chaque bot à ainsi un identifiant (m_id) et un status (m_status) accessiblement respectivement par le biais des fonctions id() et status(). La fonction is_compromised() permet de savoir si un bot est entre les mains de l'attaquant. Le bot est dit compromit par l'attaquant si et seulement si le statut du bot vaut STOPPED et a ce moment le fonctionnement du Bot est stoppé (il ne peut pas repasser à l'ennemi).

Chaque Bot peut effectuer des "actions", pour l'instant seul l'envoi d'un message est possible (par le biais de la fonction send_message()).

5 Classe Attacker

Cette classe implémente les fonctionnalitées d'un Bot de type attaquant. Elle dérive de la class Repeater (la plupart des attaques sur le botnet waledac se font en introduisant des repeaters).

```
class Attacker : public Repeater
{
private:
#ifdef THREAD_VERSION
boost::shared_ptr< boost::thread >
m_attacker_thread;
#endif

public:
Attacker();
virtual void update_rlist();
virtual void update_plist();
virtual bots_t sub_rlist();
virtual bots_t sub_plist();

virtual response_code
send_message(message_code message);

#ifdef THREAD_VERSION
virtual void execute();
virtual void start();
virtual void wait();
#endif
};
```

6 Classe Protecter

Cette classe implémente les fonctionnalitées d'un Bot de type Protecter.

```
class Protecter : public Bot
{
private:
#ifdef THREAD_VERSION
boost::shared_ptr<boost::thread>
m_protecter_thread;
#endif

public:
Protecter();
bot_t server;

virtual response_code
send_message(message_code message);
virtual void init(bot_t& server,
bots_t& plist, bots_t& rlist);

#ifdef THREAD_VERSION
virtual void execute();
        virtual void start();
        virtual void wait();
#endif
};
```

7 Classe ServerCC

Cette classe implémente les fonctionnalitées d'un Bot de type ServerCC.

```
class ServerCC : public Bot
{
private:
#ifdef THREAD_VERSION
boost::shared_ptr<boost::thread>
m_servercc_thread;
#endif

public:
ServerCC();

response_code
process_message(message_code message);
virtual response_code
send_message(message_code message);

virtual void init(bot_t& server,
bots_t& plist, bots_t& rlist);

#ifdef THREAD_VERSION
virtual void execute();
virtual void start();
virtual void wait();
#endif

};
```

8 Classe Botnet

Cette classe implémente les fonctionnalitées du Botnet.

```
class Botnet :
public boost::enable_shared_from_this< Botnet >
{
public:
Botnet(unsigned int repeaters_number,
unsigned int protecters_number,
unsigned int spammers_number,
unsigned int attackers_number);

void init();
void start();

#ifdef THREAD_VERSION
void wait();
#endif

bots_t repeaters_list();
bots_t protecters_list();
bots_t spammers_list();
bots_t attackers_list();
bot_t server();


private:
bots_t repeaters;
bots_t protecters;
bots_t spammers;
bots_t attackers;
bot_t server_cc;
};
```

On crée un botnet via le constructeur, qui prend en paramètre un nombre de répéteurs, protecteurs, spammers, attaquants, il n'y aura qu'un seul command and conquer.

On a accès à la liste des bots (bots_t repeaters, bots_t protecters, bots_t spammers, bots_t attackers, bot_t server_cc) via les fonctions repeaters_list(),protecters_list(), spammers_list(), attackers_list(), server().

9 Visualisation

Le graphe du Botnet dérive de la classe vtkMutableDirectedGraph (il s'agit d'un graphe dirigé et modifiable). Dans la version avec threads, le rafraichissement du rendu graphique est effectué toutes les 300 millisecondes (par défaut), les bots peuvent avoir effectué plus d'une action chacun (réception, envoie d'un message par exemple), dans la version sans threads ce n'est que lors des rafraichissements du rendu graphique que les bots "vivent".