



MalwareByte Challenge 2

Challenge's write-up

Wed Nov 6th, 2019



Takuya Sawada
The University of Electro-Communications, Tokyo
 @takuzoo3868
 sata3868.k.serpentis@gmail.com

Contents

1	はじめに	3
2	Analysis of mb_crackme_2.exe	3
2.1	Stage 1: Login	3
2.2	Stage 2: The Secret Console	7
2.3	Stage 3: The Color of Reverse Engineering	15
3	終わりに	17

1 はじめに

🐦@hasherazade 氏が開催した Malwarebytes CrackMe 2: try another challenge の WriteUP です。CTF より問題の目的がはっきりしているため、世の中のマルウェアへ抗う手段を模索したい方、マルウェア解析者になりたい方におすすめです。

開催時は作者の GoogleDrive から入手できましたが、現在は [hybrid-analysis](#)^{*1} にアップロードされています^{*2}。
📎**mb_crackme_2.exe** は 8.3MiB 程度の実行ファイルです。主たる解析の流れは図 1 に示した通りです。筆者は Windows10 に flare-vm^{*3} を当てた環境で問題を解いています。それでは次の節から解説をはじめます。

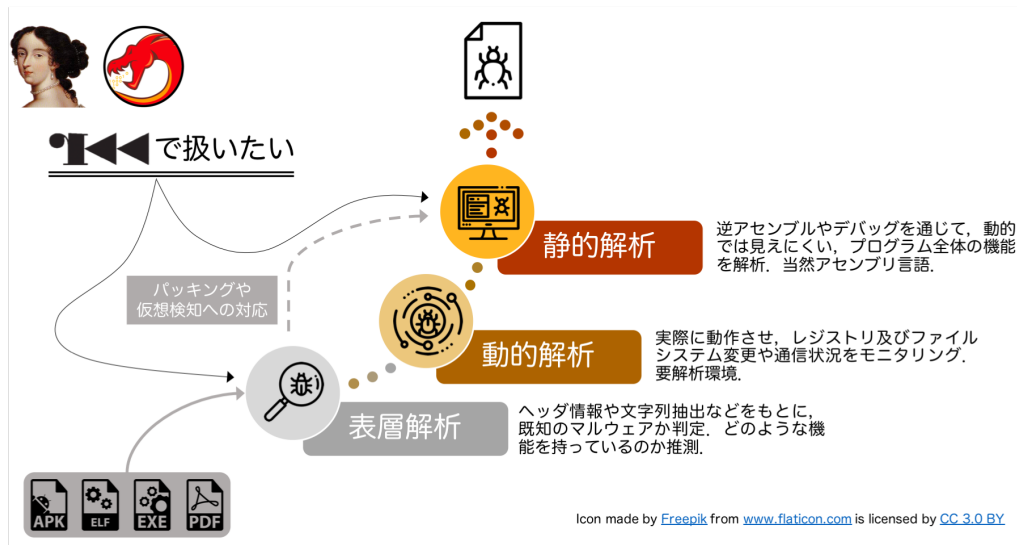


Figure1: マルウェア解析の概略

2 Analysis of mb_crackme_2.exe

2.1 Stage 1: Login

まず、ヘッダ情報を確認するとバイナリソースの大半は **.text** に集約されている事がわかります。一方で、リソースにはアイコン画像しか格納されていないことが確認できます。CFF Explorer^{*4} を利用すると図 2 のように確認できます。8.3MiB の大半が **.text** にあるということは、Hex editor を使い目的の情報を目 grep するには辛すぎます。string などの文字列抽出ツールを使い、長い文字列を抜き出し特徴を探ってみます。

^{*1} <http://tinyurl.com/ybuf6z2h>

^{*2} 検体の入手には認証が必要ですのでご注意ください

^{*3} マルウェア解析用に FireEye 社がカスタマイズした Windows 用ツール群

^{*4} ファイル識別、アドレスの変換、依存関係のスキャン、インポートされた関数を調べる際に便利な PE エディタです

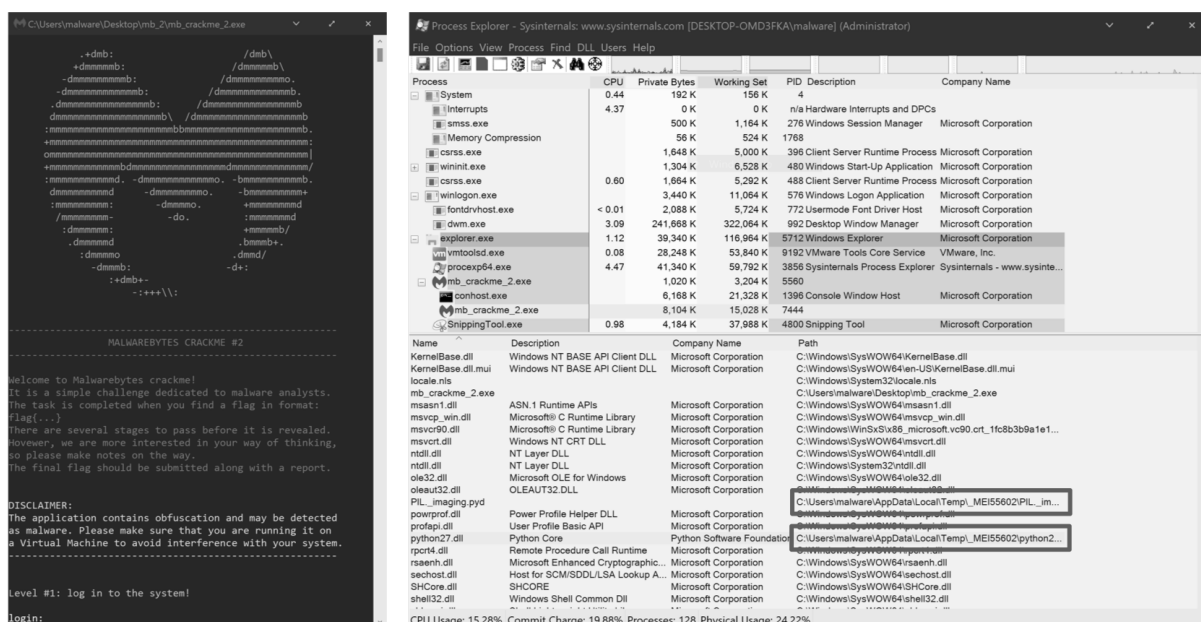


Figure3: 起動した様子とプロセスの監視

難読化された Python をアンパックするために、python 用のアンパッカーである `python-exe-unpacker`^{*5} を使います。吐き出された中身を見ると、`another` というコンパイル済みのファイルを確認できます。uncompyle6 を使って逆コンパイルを試みます^{*6}が、`ImportError: Unknown magic number` となり、マジックナンバーが消えている事がわかりました。この記事^{*7}を参考にしたところ、PyInstaller を使った main スクリプトの圧縮では、時折マジックナンバーを消すことがあるので、`0x03 0xF3 0x0D 0x0A 0x00 0x00 0x00 0x00` へ書き換える必要があるとわかりました。再度逆コンパイルを試すと `another.py` という 300 行程度のソースコードを取得できました。

アンパック

```
PS C:\Users\malware\Desktop\mb_2> .\python_exe_unpack.py -i .\mb_crackme_2.exe
[*] On Python 2.7
[*] Processing .\mb_crackme_2.exe
[*] Pyinstaller version: 2.1+
[*] This exe is packed using pyinstaller
[*] Unpacking the binary now
[*] Python version: 27
[*] Length of package: 8531014 bytes
[*] Found 931 files in CArchive
[*] Beginning extraction...please standby
[*] Found 440 files in PYZ archive
[*] Successfully extracted pyinstaller exe.
```

これで Cracking の準備が整いました。ログインに関連する箇所は容易に見つかります。

```
def main():
    key = stage1_login()
    if not check_if_next(key):
        return
```

^{*5} <https://github.com/countercept/python-exe-unpacker>

^{*6} このツールは拡張子判定に時間がかかるため、予め.pyc へ変更しておくといいです。

^{*7} <https://0xec.blogspot.com/2017/12/reversing-pyinstaller-based-ransomware.html>

```
def stage1_login():
    show_banner()
    print colorama.Style.BRIGHT + colorama.Fore.CYAN
    print 'Level #1: log in to the system!'
    print colorama.Style.RESET_ALL
    login = raw_input('login: ')
    password = getpass.getpass()
    if not (check_login(login) and check_password(password)):
        print 'Login failed. Wrong combination username/password'
        return
    PIN = raw_input('PIN: ')
    try:
        key = get_url_key(int(PIN))
    except:
        print 'Login failed. The PIN is incorrect'
        return
    else:
        if not check_key(key):
            print 'Login failed. The PIN is incorrect'
            return
    return key
```

stage1_login では username, password, PIN を要求しているようです。また、これらをチェックする関数も見つかりました。ユーザー名はただの平文で hackerman, パスワードについては MD5 でハッシュ化されていますが⁴, オンラインにある decryptor tool を使えば Password123 とわかります。PIN については少々複雑です。get_url_key に PIN を渡してキーを生成し、次に check_key で検証しています。Python の乱数生成モジュールを使って PIN をシードとした 0 から 9 の数字からなる 32 桁の数値キーを生成しています。検証では md5 のハッシュを使っているようです。PIN などで 4 桁の可能性が高いだろうと予想して、総当たりでキーを生成し MD5 ハッシュを検証することになります。

```
def check_key(key):
    my_md5 = hashlib.md5(key).hexdigest()
    if my_md5 == 'fb4b322c518e9f6a52af906e32aee955':
        return True
    return False

def check_login(login):
    if login == 'hackerman':
        return True
    return False

def check_password(password):
    my_md5 = hashlib.md5(password).hexdigest()
    if my_md5 == '42f749ade7f9e195bf475f37a44cafcbb':
        return True
    return False

def get_url_key(my_seed):
    random.seed(my_seed)
    key = ''
    for i in xrange(0, 32):
        id = random.randint(0, 9)
        key += str(id)

    return key
```

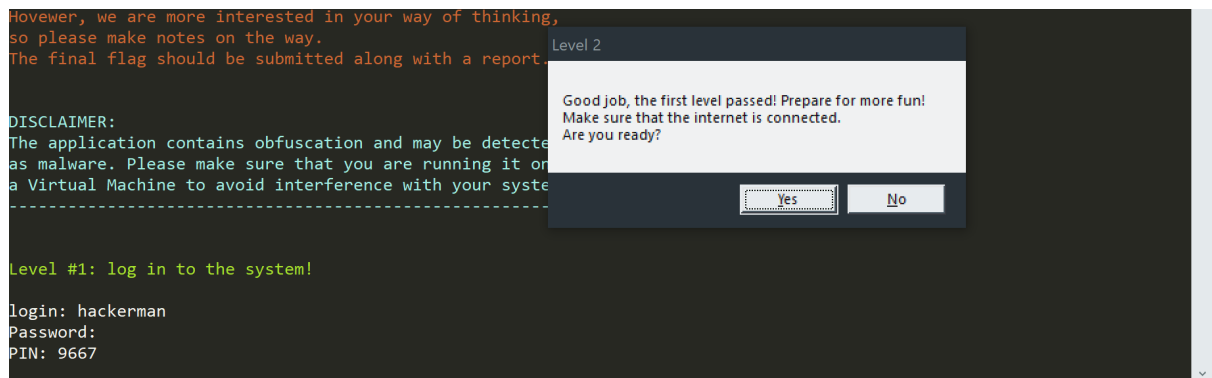
作成したソルバーと実行結果

```
import another

for i in xrange(1000000):
    key = another.get_url_key(i)
    if another.check_key(key):
        print 'PIN: ', i
        print 'key'

PS C:\Users\malware\Desktop\mb_2> pip install --user pycrypto Pillow colorama
PS C:\Users\malware\Desktop\mb_2> python .\solve.py
PIN: 9667
key: 95104475352405197696005814181948
```

これで PIN も 9667 とわかりました。よって、ログインは突破できました 🎉。



2.2 Stage 2: The Secret Console

再度ソースコードを見ていきます。check_if_next でダイアログを表示し、Level2 への準備ができているかどうかの確認を行っています。先程の PIN を利用したキーは decode_and_fetch_url へ渡され、AES で暗号化された URL を復号化しリンク先から何かしらのコンテンツを取得しているように思えます。URL を確認するために another.py へ一行追加して実行してみました。取得できた 📷 データを図 4 に示します。

```
def main():
    key = stage1_login()
    if not check_if_next(key):
        return
    content = decode_and_fetch_url(key)
    if content is None:
        print 'Could not fetch the content'
        return -1
    decdata = get_encoded_data(content)
    if not is_valid_payl(decdata):
        return -3
    print colorama.Style.BRIGHT + colorama.Fore.CYAN
    print 'Level #2: Find the secret console...'
    print colorama.Style.RESET_ALL
    load_level2(decdata, len(decdata))
```

```
def decode_and_fetch_url(key):
    try:
        encrypted_url =
'\xa6\xfa\x8f0\xba\x7f\x9d\xe2c\x81`\xf5\xd5\xf6\x07\x85\xfe[hr\xd6\x80?U\x90\x89)\xd1\xe9
\x0<\xfe'
        aes = AESCipher(bytearray(key))
        output = aes.decrypt(encrypted_url)
        full_url = output
        print "DEBUG : URL fetched is : %s " % full_url #added from original code
        content = fetch_url(full_url)
    except:
        return

    return content
```

```
login: hackerman
Password:
PIN: 9667
DEBUG : URL fetched is : https://i.imgur.com/dTHXed7.png
```



Figure4: プログラムが取得していた画像データ

一見するとただの PNG 形式ですが、明らかに怪しいノイズデータとなっています。無害なファイル形式を装ってコンテンツをローカルへ保存する手法はネットワークの監視をすり抜ける常套手段でもあります。今回の例であれば、監視ツールから見ると imgur からただ画像を取得したと判断されたのだと思います。特に PNG 形式は可逆圧縮なのでコンテンツの隠れ蓑として優秀であると言えます^{*8}。関数 `get_encoded_data` では PNG 形式の RGB 成分からバイトコードへ変換されています。また、フォーマットを解析して、バイトコードからコンテンツを展開するために PIL を使った関数は `is_valid_payl` でした。関数内でチェックしている 23117 と 17744 は、16 進数へ変換してやると MZ や PE のマジックナンバーであることに気が付きました。つまり、この関数はデコードしたデータを PE 形式などの実行可能ファイルへ変換していると推察できます。実行ファイルを抽出するために `another.py` を再度書き換えます。main では次に `load_level12` ヘデコードしたデータを渡しています。

*8 CTF で言えば意図的に情報を隠すステガノグラフィとして一大分野になっていますよね...?


```
def is_valid_payl(content):
    if get_word(content) != 23117:
        return False
    next_offset = get_dword(content[60:])
    next_hdr = content[next_offset:]
    if get_dword(next_hdr) != 17744:
        return False
    return True
```

```
def load_level2(rawbytes, bytesread):
    try:
        if prepare_stage(rawbytes, bytesread):
            return True
    except:
        return False

def prepare_stage(content, content_size):
    virtual_buf = kernel_dll.VirtualAlloc(0, content_size, 12288, 64)
    if virtual_buf == 0:
        return False
    res = memmove(virtual_buf, content, content_size)
    if res == 0:
        return False
    MR = WINFUNCTYPE(c_uint)(virtual_buf + 2)
    MR()
    return True
```

ソースコードを読むと、この関数は、まずはじめに VirtualAlloc を使用して、コードを格納するための十分な領域を確保しています。第四引数の 64 は、PAGE_EXECUTE_READWRITE を意味するので読み書き可能・実行可能な権限で領域であるとわかります。対象は memmove 関数によって書き込みが実行されるといった内容です。prepare_stage へ数行追加してコンテンツのダンプファイルを取得してみましょう。

ダンプファイルの取得

```
def prepare_stage(content, content_size):
    with open("dumped_payload.dll", "wb") as f: # added from original code
        f.write(content[:content_size])
        print "DEBUG : File dumped in dumped_payload.dll"
    virtual_buf = kernel_dll.VirtualAlloc(0, content_size, 12288, 64)
    # ...
```


```
PS C:\Users\malware\Desktop\mb_2> python .\another.py
Level #2: Find the secret console...
```

```
DEBUG : File dumped in dumped_payload.dll
```

```
PS C:\Users\malware\Desktop\mb_2> trid.exe .\dumped_payload.dll
```

```
TrID/32 - File Identifier v2.24 - (C) 2003-16 By M.Pontello
Definitions found: 11384
Analyzing...
```

```
Collecting data from file: .\dumped_payload.dll
52.9% (.EXE) Win32 Executable (generic) (4508/7/1)
23.5% (.EXE) Generic Win/DOS Executable (2002/3)
23.5% (.EXE) DOS Executable Generic (2000/1)
```

ディレクトリに  dumped_payload.dll が保存され、実行形式であることを確認できたので、Ghidra^{*9}を使って中身を解析していきます。Ghidra は DLL entry point へ自動遷移してくれないので^{*10}、SymbolTree から Export で Entry を探るか、SymbolTable を展開して entry でフィルタをかけるなどして、0x100086ac へ移動します。ページ数の都合上省きますが、DllMainCTRStartup や他の関数のアドレス配置^{*11}から FUN_10001170 であると仮定しました。

^{*9} NSA が公開した GUI ベースのリバースエンジニアリングツール。公式 FAQ によれば Gee-druh と発音します。IDA と違い無料でデコンパイラを使えます。まあ Retdec がありますが

^{*10} 使い始めたばかりで設定をよくわかってないため、こうすれば便利だとかあったらご教授ください

^{*11} FUN_10001170 以外の関数は、entry point から遷移する FUN_10008579 に近いアドレス配置のため静的にリンクされた単一モジュールと推測しました

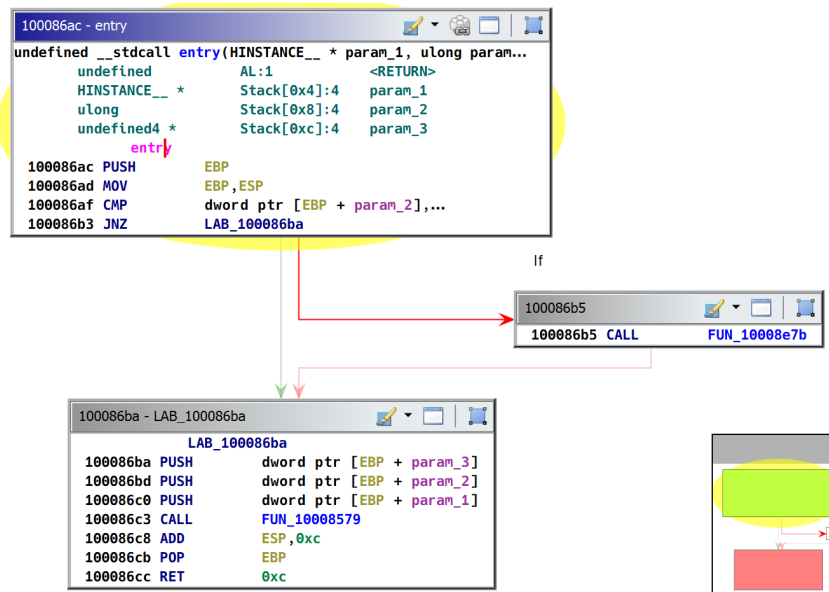


Figure5: DllEntryPoint function

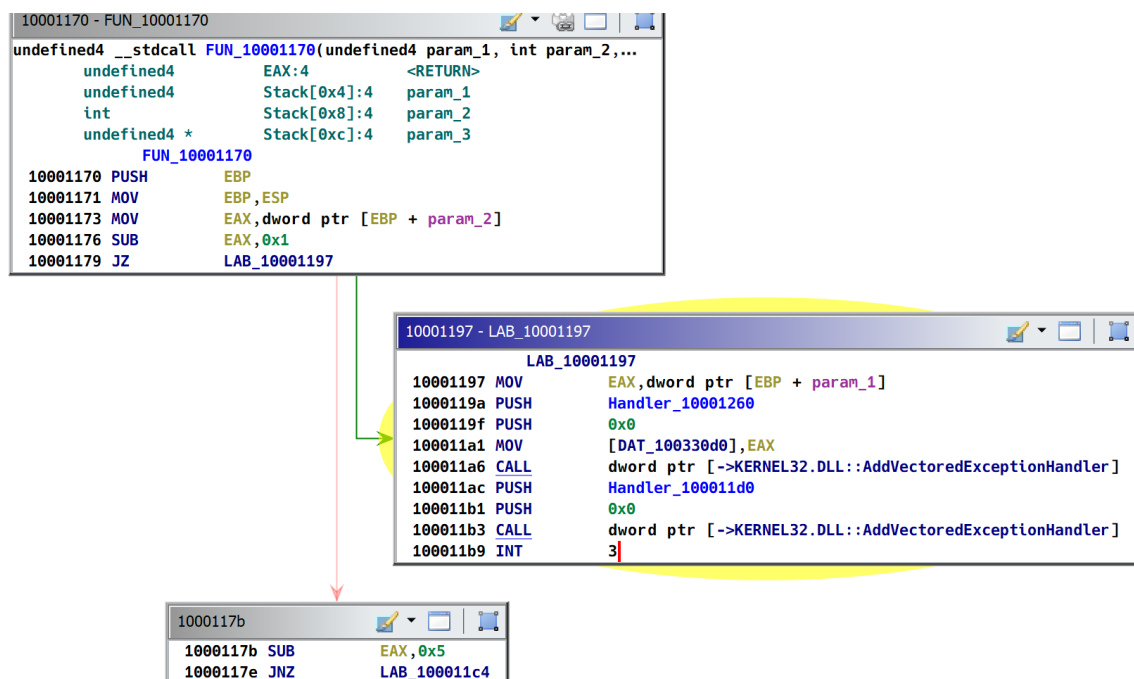


Figure6: DllMain function

最初のセクションでは `AddVectoredExceptionHandler` を呼び出して例外ハンドラを2つ設定しています。ここで面白い処理をしているのが `0x100011b9` にある `INT 3` です。割り込みにより自発的に例外を発生させ、例外ハンドラへ遷移するように仕掛けてあります。解析により2つハンドラは `Handler_10001260`, `Handler_100011d0` であると判明しています。`Handler_10001260` では、`python27.dll` のロードを検出し、変数 `mb_chall` にプロセスIDを格納しています。`EXCEPTION_CONTINUE_SEARCH` で最終的には次のハンドラへ遷移します。`Handler_100011d0` では、`python27.dll` の

ロードを検出し、EIP^{*12}を1ずつ或いは6ずつ増やし、プログラムを通常実行するようOSへ指示しています。DllMainでは、先程述べたINT 3例外により、例外ハンドラを呼び出すことでEIPにはINT 3の先頭である0x100011B9が格納されます。これはマルウェアにおけるアンチデバッグの一種と考えられます。ざっくり言うと、プロセスにアタッチしているデバッガは、このINT 3は例外ハンドラへ渡されたわけではなく、作者が意図的に設定した命令であるとみなしてスルーしてしまいます。Pythonがロードされていないと判定した場合、先程の例外ハンドラはEIPを1増やし、メッセージダイアログを表示するFUN_100010F0へ移ります。一方でpython27.dllがロードされている場合には、FUN_100010F0をスキップしてFUN_00010D0を呼び出します。

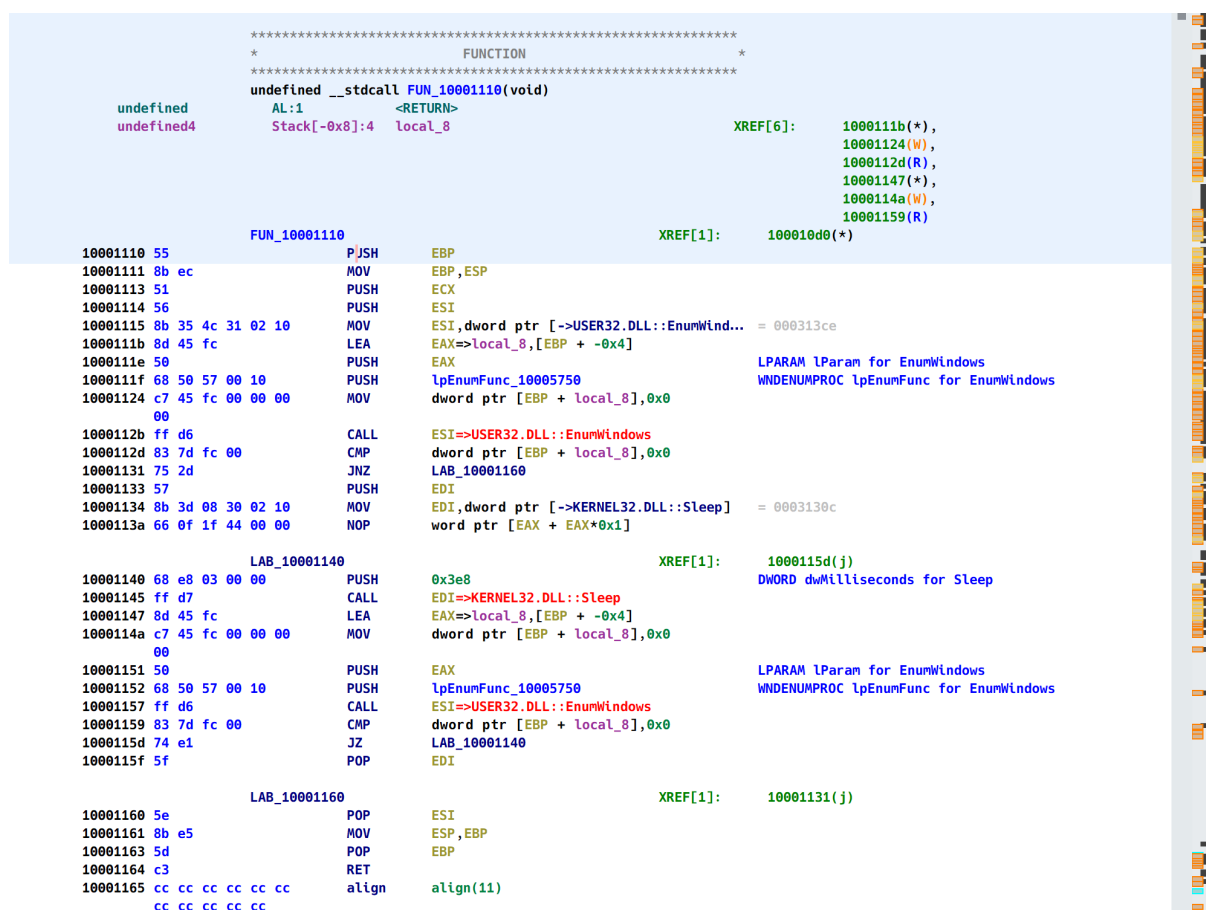


Figure7: MainThread function

*12 Extended Instruction Pointer Register

例外ハンドラ箇所の clang っぽい書き方

```

LONG WINAPI Handler_10001260(struct _EXCEPTION_POINTERS *ExceptionInfo) {
    char Value[0x104];
    memset(Value, 0, sizeof(Value));
    if (GetModuleHandle("python27.dll") != 0) {
        _itoa_s(GetCurrentProcessId(), Value, sizeof(Value), 10);
    }
    SetEnvironmentVariable("mb_chall", Value);
    return EXCEPTION_CONTINUE_SEARCH; // return 0
}

LONG WINAPI Handler_10001d0(struct _EXCEPTION_POINTERS *ExceptionInfo) {
    char Buffer[0x104];
    memset(Buffer, 0, sizeof(Buffer));
    PCONTEXT pctx = ExceptionInfo->ContextRecord; // ebx
    DWORD delta = 1; // edi
    if (GetEnvironmentVariable("mb_chall", Buffer, sizeof(Buffer))) {
        if (sub_1000E409(Buffer) == GetCurrentProcessId()) // probably atoi()
            delta = 6;
    }
    pctx->Eip += delta; // EIP is at offset 0xB8 of CONTEXT
    return EXCEPTION_CONTINUE_EXECUTION; // return -1
}

```

では FUN_00010D0 の中身を追っていきます。FUN_00059D0 で `CreateThread` を呼び出してバックグラウンドスレッドを開始しています。その後に、`WaitForSingleObject` を使って、子スレッドの完了まで待機状態になる様です。スレッドの開始点は `push FUN_10001110` でしょう。これも推測ですが、IDA を使っていた人は `StartAddress` へ変換されていたので合っていると思います。コールバック関数である `lpEnumFunc_10005750` にて起動中のウィンドウを列挙し、`dwMilliseconds` に設定されている間隔で監視を繰り返します。起動しているウィンドウのタイトルは `WM_GETTEXT hWnd` へ渡され、結果は `lParam` へスタックされます。以降解析を進めると LAB_10005823 付近から `Notepad` や `secret_console` など特徴的な文字列が出現します^{*13}。WM_SETTEXT を渡した `SendMessageA` API を使って、先程の文字列が両方存在する場合、ウィンドウのタイトルを **Secret Console is waiting for the commands...** 書き換えます。更に `ShowWindow` API を使って書き換えたウィンドウを最前面に持ってくるといった処理をする様です。

10005797 e8 24 3b 00 00	CALL	_memset	void * _memset(void * _Dst, int _Val, size_t _Size)
1000579c 83 c4 0c	ADD	ESP,0xc	
1000579f 8d 85 bc fe ff ff	LEA	EAX=>local_148,[0xfffffeb c + EBP]	
100057a5 50	PUSH	EAX	LPARAM lParam for SendMessageA
100057a6 68 04 01 00 00	PUSH	0x104	WPARAM wParam for SendMessageA
100057ab 6a 0d	PUSH	0xd	UINT Msg for SendMessageA
100057ad 57	PUSH	EDI	HWND hWnd for SendMessageA
100057ae ff 15 48 31 02 10	CALL	dword ptr [->USER32.DLL::SendMessageA]	
100057b4 85 c0	TEST	EAX,EAX	

Figure8: SendMessageA call in MainThread function

^{*13} 通常であれば文字列検索で先に見つかるかと思いますが、Main 関数から大まかな仕組みをたどるために敢えて解説では後の方になりました

```

1000582a  SUB     ECX,EDX
1000582c  LEA     EAX,local_148,[0xfffffeb4 + EBP]
10005832  PUSH    ECX
10005833  PUSH    EAX
10005834  LEA     ECX,local_160,[0xfffffea4 + EBP]
1000583a  CALL    FUN_10004630
1000583f  PUSH    0x0
10005841  PUSH    s_Notepad_100233bc
10005846  LEA     ECX,local_160,[0xfffffea4 + EBP]
1000584c  MOV     dword ptr [EBP + local_8],0x0
10005853  CALL    FUN_100051a0
10005858  CMP     EAX,-0x1
1000585b  JZ      LAB_100058b9

1000585d  PUSH    0x0
1000585f  PUSH    s_secret_console_100233c4
10005864  LEA     ECX,local_160,[0xfffffea4 + EBP]
1000586a  CALL    FUN_100051a0
1000586f  CMP     EAX,-0x1
10005872  JZ      LAB_100058b9

```

Figure9: "Notepad" and "secret_console"

```

100058a2  8d 45 c0      LEA     EAX,local_44,[EBP + -0x40]
100058a5  50           PUSH    EAX
100058a6  56           PUSH   ESI
100058a7  6a 0c        PUSH    0xc
100058a9  57           PUSH    EDI
100058aa  ff 15 48 31 02 10  CALL    dword ptr [->USER32.DLL::SendMessageA]
100058b0  6a 05        PUSH    0x5
100058b2  57           PUSH    EDI
100058b3  ff 15 44 31 02 10 CALL    dword ptr [->USER32.DLL::ShowWindow]

LPARAM lParam for SendMessageA
WPARAM wParam for SendMessageA
UINT Msg for SendMessageA
HWND hWnd for SendMessageA

int nCmdShow for ShowWindow
HWND hWnd for ShowWindow

```

Figure10: SendMessageA API を使ったウィンドウタイトルの上書き

```

LAB_100058b9  XREF[2]: 1000585b(j), 10005872(j)
100058b9  6a 00      PUSH    0x0
100058bb  8d 45 c0      LEA     EAX,local_44,[EBP + -0x40]
100058be  50           PUSH    EAX
100058bf  8d 8d a4 fe ff ff LEA     ECX,local_160,[0xfffffea4 + EBP]
100058c5  e8 d6 f8 ff ff CALL    FUN_100051a0
100058ca  83 f8 ff     CMP     EAX,-0x1
100058cd  74 4e        JZ      LAB_1000591d
100058cf  8d 85 a0 fe ff ff LEA     EAX,local_164,[0xfffffea0 + EBP]
100058d5  c7 85 a0 fe ff ff MOV     dword ptr [local_164 + EBP],0x0
100058df  50           PUSH    EAX
100058e0  57           PUSH    EDI
100058e1  ff 15 58 31 02 10 CALL    dword ptr [->USER32.DLL::GetWindowThreadProcessId]
100058e7  68 d4 33 02 10 PUSH    s_...waiting_for_the_command_100233d4
100058ec  ff b5 a0 fe ff ff PUSH    dword ptr [local_164 + EBP]
100058f2  b9 00 32 03 10 MOV     ECX,DAT_10033200
100058f7  e8 d4 d6 ff ff CALL    FUN_10002fd0
100058fc  50           PUSH    EAX
100058fd  e8 4e bf ff ff CALL    FUN_10001850
10005902  50           PUSH    EAX
10005903  e8 98 c9 ff ff CALL    FUN_100022a0
10005908  83 c4 0c     ADD     ESP,0xc
1000590b  ff b5 9c fe ff ff PUSH    dword ptr [local_168 + EBP]
10005911  68 c0 34 00 10 PUSH    lpEnumFunc_100034c0
10005916  57           PUSH    EDI
10005917  ff 15 54 31 02 10 CALL    dword ptr [->USER32.DLL::EnumChildWindows]

LPDWORD lpdwProcessId for GetWindowThreadProcessId
HWND hWnd for GetWindowThreadProcessId

= ": waiting for the command"
= ??
int * FUN_10002fd0(void * this, undefined4 param_1)
int * FUN_10001850(int * param_1, char * param_2)
int * FUN_100022a0(int * param_1)
LPARAM lParam for EnumChildWindows
WNDENUMPROC lpEnumFunc for EnumChildWindows
HWND hWndParent for EnumChildWindows

```

Figure11: call EnumChildWindows API

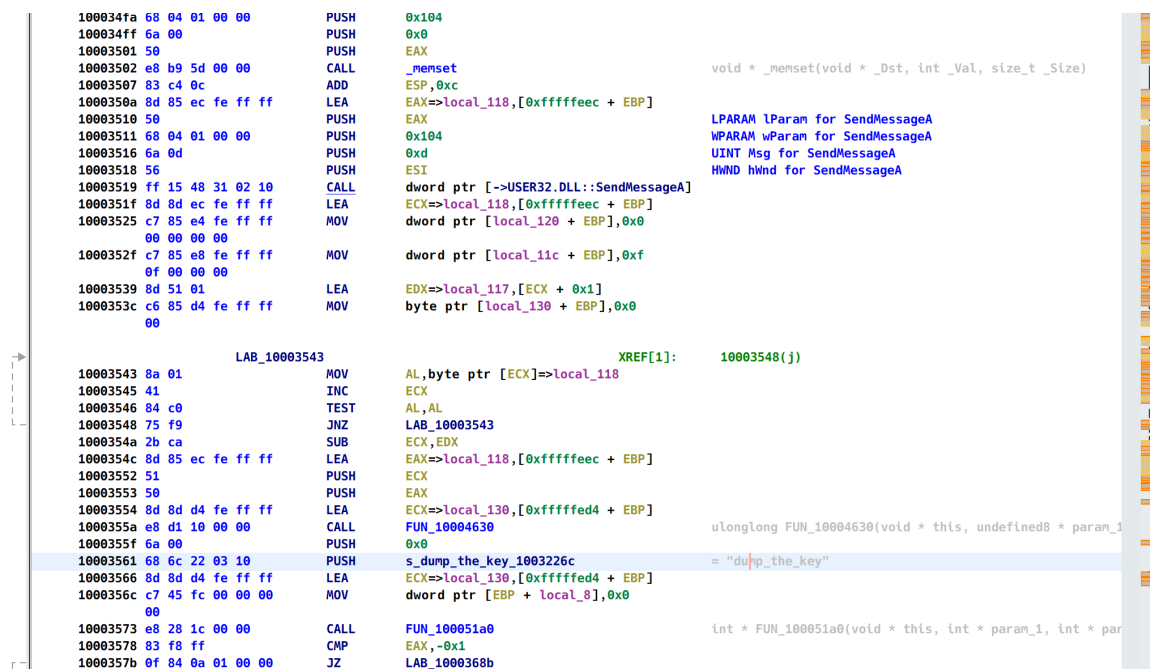


Figure12: "dump_the_key"

処理は EnumChildWindows API を使って子ウィンドウの列挙へと移ります。なので、次に注目すべきは lpEnumFunc.100034c0 内部となります。すぐに dump_the_key という怪しい文字列が見つかるはずですが。周辺の処理をじっくり眺めると、SendMessageA API を叩いてサブウィンドウの内容を取得し、dump_the_key と合致するコンテンツを検索します。発見した場合は文字列そのものを引数として decrypt_buffer を呼び出します。その後、actxprxy.dll がメモリにロードされ、先程復号化処理を行ったデータを actxprxy.dll の先頭から 4096byte に書き込み、スレッド処理及び DllEntryPoint も終了し、プロセス制御を mb.crackme.2.exe へ戻します。したがって、要点は以下のとおりです。

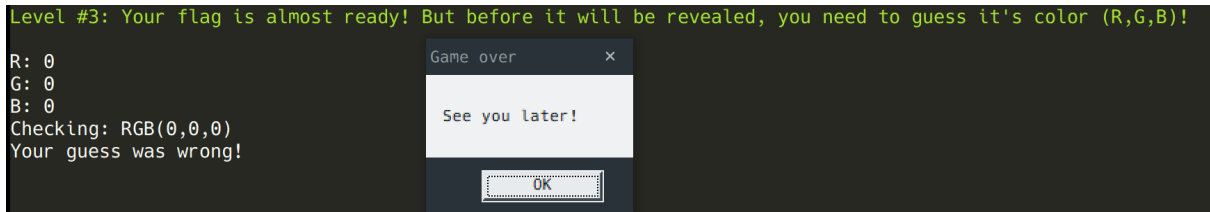
- ウィンドウタイトルに secret_console が表示されるようなメモ帳を展開する
- メモ帳内部 (サブウィンドウ) で dump_the_key と入力

mb.crackme.2.exe を Level2 まで進めた上で、secret_console.txt のような空ファイルをメモ帳で開き、dump_the_key と入力してみます。無事突破できました👏。



2.3 Stage 3: The Color of Reverse Engineering

次の問題は色に関係している様です。適当に入力しても駄目なので再度ソースコードを読みます。



```
def main():
    # ...
    load_level2(decdata, len(decdata))
    user32_dll.MessageBoxA(None, 'You did it, level up!', 'Congrats!', 0)
    try:
        if decode_pasted() == True:
            user32_dll.MessageBoxA(None,
                'Congratulations! Now save your flag and send it to Malwarebytes!', 'You solved it!', 0)
            return 0
            user32_dll.MessageBoxA(None, 'See you later!', 'Game over', 0)
    except:
        print 'Error decoding the flag'
```


```
def dextr_data(data, key):
    maxlen = len(data)
    keylen = len(key)
    decoded = ''
    for i in range(0, maxlen):
        val = chr(ord(data[i]) ^ ord(key[i % keylen]))
        decoded = decoded + val

    return decoded

def decode_pasted():
    my_proxy = kernel_dll.GetModuleHandleA('actxprxy.dll')
    if my_proxy is None or my_proxy == 0:
        return False
    char_sum = 0
    arr1 = my_proxy
    str = ''
    while True:
        val = get_char(arr1)
        if val == '\x00':
            break
        char_sum += ord(val)
        str = str + val
        arr1 += 1

    print char_sum
    if char_sum != 52937:
        return False
    colors = level3_colors()
    if colors is None:
        return False
    val_arr = zlib.decompress(base64.b64decode(str))
    final_arr = dextr_data(val_arr, colors)
    try:
        exec final_arr
    except:
        print 'Your guess was wrong!'
        return False

    return True
```

main で呼び出している decode_pasted に着目すると、Level2 で最後に  actxprxy.dll を使い復号化したバッファを読み込んでいる様です。

1. base64 で復号化
2. zlib でデータを解凍
3. ユーザーの入力した RGB 値と XOR
4. exec 関数の使用

大まかな流れは把握できたので、final_arr の中身を探します。Level2 まで進めた状態で x32dbg ヘアタッチします^{*14}。その状態でメモリマップを確認すると上書きされた actxprxy.dll が見つかるのでダンプします。HexEditor などで確認すると図 13 のようなデータを確認できました。python を使ってダンプしたバイナリファイルをよしなに変換すると復号化できてしまいました。

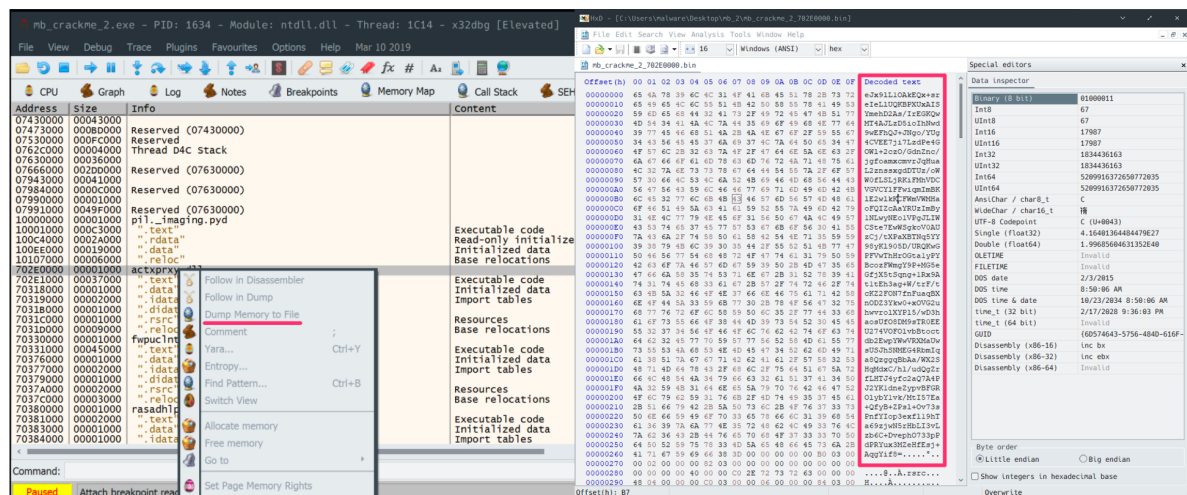


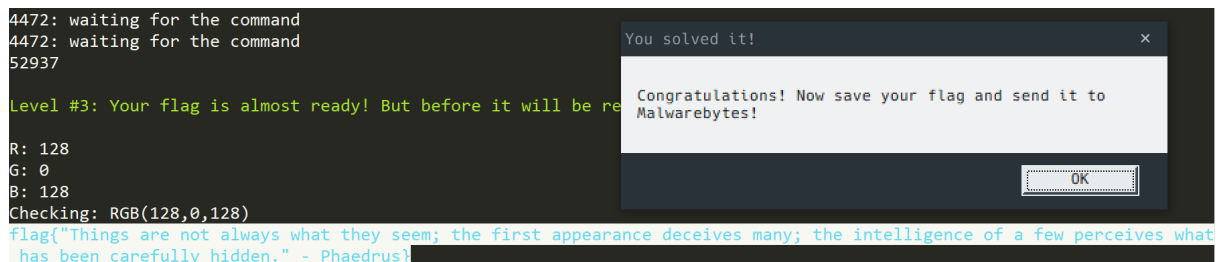
Figure13: ダンプした actxprxy.dll の中身

^{*14} PID がわからない場合は procexp などを使って調べます

XOR を使ったデコード

```
>>> import zlib
>>> from Crypto.Cipher import XOR
>>> crypted = zlib.decompress(open('702E0000.bin', 'rt').read()).decode('base64'))
>>> len(crypted)
1134
>>> crypted[:8].encode('hex')
'e465e6a070f2e96e'
>>> data = XOR.new('\x80\x00\x80').decrypt(crypted)
>>> print data
def print_flag():
    flag_hex = (
        0x73, 0x75, 0x72, 0x64, 0x65, 0x61, 0x68, 0x50, 0x20, 0x2D, 0x20, 0x22, 0x2E, 0x6E,
        0x65, 0x64, 0x64, 0x69, 0x68, 0x20, 0x79, 0x6C, 0x6C, 0x75, 0x66, 0x65, 0x72, 0x61,
        0x63, 0x20, 0x6E, 0x65, 0x65, 0x62, 0x20, 0x73, 0x61, 0x68, 0x20, 0x74, 0x61, 0x68,
        0x77, 0x20, 0x73, 0x65, 0x76, 0x69, 0x65, 0x63, 0x72, 0x65, 0x70, 0x20, 0x77, 0x65,
        0x66, 0x20, 0x61, 0x20, 0x66, 0x6F, 0x20, 0x65, 0x63, 0x6E, 0x65, 0x67, 0x69, 0x6C,
        0x6C, 0x65, 0x74, 0x6E, 0x69, 0x20, 0x65, 0x68, 0x74, 0x20, 0x3B, 0x79, 0x6E, 0x61,
        0x6D, 0x20, 0x73, 0x65, 0x76, 0x69, 0x65, 0x63, 0x65, 0x64, 0x20, 0x65, 0x63, 0x6E,
        0x61, 0x72, 0x61, 0x65, 0x70, 0x70, 0x61, 0x20, 0x74, 0x73, 0x72, 0x69, 0x66, 0x20,
        0x65, 0x68, 0x74, 0x20, 0x3B, 0x6D, 0x65, 0x65, 0x73, 0x20, 0x79, 0x65, 0x68, 0x74,
        0x20, 0x74, 0x61, 0x68, 0x77, 0x20, 0x73, 0x79, 0x61, 0x77, 0x6C, 0x61, 0x20, 0x74,
        0x6F, 0x6E, 0x20, 0x65, 0x72, 0x61, 0x20, 0x73, 0x67, 0x6E, 0x69, 0x68, 0x54, 0x22 )
    flag_str = ""
    for i in flag_hex:
        flag_str = chr(i) + flag_str
    init()
    print(Style.BRIGHT + Back.MAGENTA + "flag{" + flag_str + "}" + (Style.RESET_ALL))
print_flag()
```

flag という文字列が怪しいので、試しにスクリプトを実行すると flag を取得できました 🎉. `flag{"Things are not always what they seem; the first appearance deceives many; the intelligence of a few perceives what has been carefully hidden." - Phaedrus}` プラトンのパイドロスから引用するとは洒落が効いてますね。これで正解ですが RGB が何だったのかはターミナルカラーの MAGENTA がヒントでした。紫色の RGB を入力すると正解みたいです。正攻法が気になります。お疲れ様でした。



```
4472: waiting for the command
4472: waiting for the command
52937
Level #3: Your flag is almost ready! But before it will be ready you need to solve a puzzle.
R: 128
G: 0
B: 128
Checking: RGB(128,0,128)
flag{"Things are not always what they seem; the first appearance deceives many; the intelligence of a few perceives what has been carefully hidden." - Phaedrus}
```

3 終わりに

いかがだったでしょうか。いくつかのテクニックは実際にマルウェアを解析する際に役立つはずです。もし興味が有余でしたらこちらのサイトなどを訪れてみてください。日夜ユーザー同士が研鑽を積んでいます。昨今、何かと逮捕案件が多くエンジニア界隈が萎縮する傾向にあります。お互いを理解するために少しでも歩み寄りが必要なのではと自分は思います。最後に作家 Raymond Chandler 氏の言葉をお借りして結びの言葉とします。

The law isn't justice. It's a very imperfect mechanism. If you press exactly the right buttons and are also lucky, justice may show up in the answer. A mechanism is all the law was ever intended to be.

— Raymond Chandler