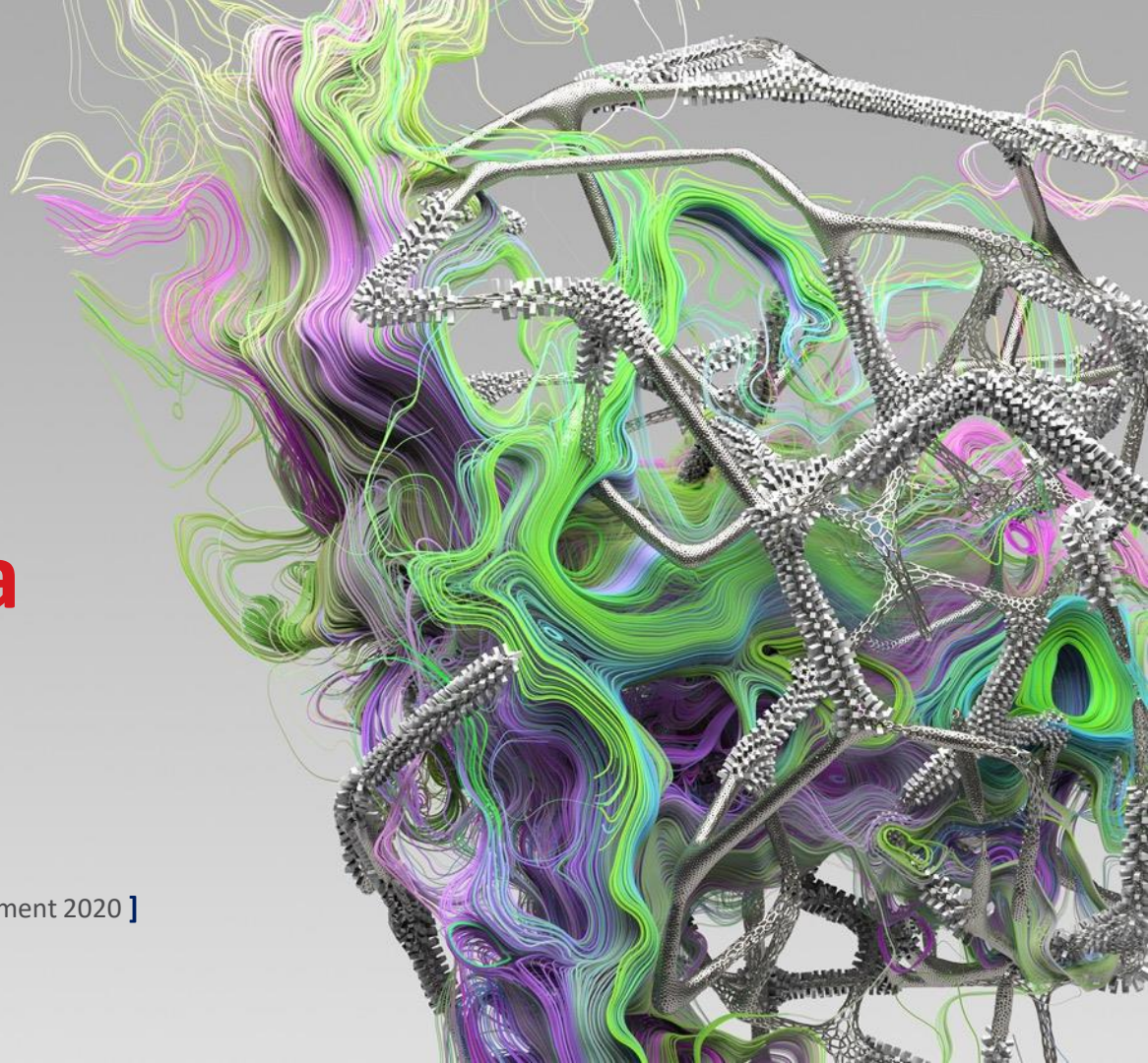
TREND MICRO

An Exploratory Endeavor in the Reverse Engineering of a Multi-platform Compiler



[Mohamad Mokbel | @MFMokbel | Living Document 2020]

May 29, 2020 - Toronto, Ontario

Biography

- Senior Security Researcher at Trend Micro
  - Member of the Digital Vaccine (DV) Lab
- Interests:
  - RE, Malware Research,
  - IDS/IPS,
  - C++, Compilers & Software Performance Analysis,
  - Exotic Communication Protocols

Introduction

- Reversing native Programming Languages (PL)
  - Phases, processes, libraries, optimizations, file formats, code generation, OS dependencies
  - Malware PL: x86 Assembly, C/C++, Objective-C, Delphi…
    - Different runtime libraries
    - Statically linked libraries (identification problem: libraries & compiler versions.)
  - The Go PL by Google required new research to be carried out to help RE Go binaries
    - Multi-platform; statically linked; no external dependencies; Go-specific metadata

Talk Layout

- PureBasic (PB)
  - The Language
  - Examples
  - Compilers, libraries
- Thesis of the Talk
  - The focus is on Windows OS
- Parsing libraries proprietary file format
  - Thought process behind it; problems encountered, and how to avoid them
- Case Studies
- Demo: Parser and IDA Pro Plugin

PureBasic – The Language

- Produces **native** code for Windows, MacOS and Linux platforms
- Extensive library support
  - Audio, Gaming, 2D/3D, DB, Networking (HTTP, FTP, SMTP), Parsing (XML/JSON), Regex, File System, Memory, Compression, GUI,…
- High level language with inline assembly support for both 32 and 64 bit archs. Uses **fasm** assembler for Win/Linux & **Yasm** for OS X
  - You can use pointers with memory access
- Support for calling into OS native shared libraries functions
  - CallFunction()/CallCFunction()
- Procedures, Structures, Interfaces with basic Inheritance (Extends)

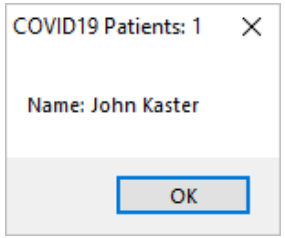
PureBasic – Example

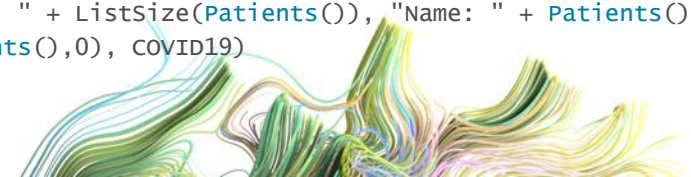
```
Structure COVID19
  Name.s
  Dead.b
  DateInfected.s
  Gender.s
  Age.i
  Symptoms.s
EndStructure

NewList Patients.COVID19()
AddElement(Patients())
; you can use With : EndWith to simplify access to struct members
Patients()\Name = “John Kaster“ ; fictional persona
Patients()\Age = 5
Patients()\Dead = true
Patients()\DateInfected = "April 03, 2020"
Patients()\Gender = “Male"
Patients()\Symptoms = “severe pneumonia, fever, headache"

SelectElement(Patients(),0)
MessageRequester("COVID19 Patients: " + ListSize(Patients()), "Name: " + Patients()\Name, #PB_MessageRequester_Ok)
ClearStructure(SelectElement(Patients(),0), COVID19)
```

- Memory
  - ClearStructure
- LinkedList
  - NewList
  - AddElement
  - SelectElement
- Requester
  - MessageRequester





PureBasic – Compilation Components (\Compilers)

- **FAsm.exe**: Flat assembler for x86 processors (32/64 bit archs). Assembly engine. It has a macroinstruction language. produces object file (*PureBasic.obj*). **COFF format**.
- **polink.exe**: Linker by Pelle Orinius, for linking generated object files and producing executable file.
- **polib.exe**: Library Manager by Pelle Orinius. To build import libraries, extract object files from a given library, list or delete.
- **porc.exe**: Resource Compiler by Pelle Orinius. For creating resource ".res" files.

pbcompiler.exe: This is the main PureBasic compiler. Parses PB source code files and emits x86 (32/64 bits) FAsm assembly instructions (PureBasic.asm). It is the orchestrator that leads to the generation of the final executable.

PB Compilation Phases

example.pb

```
MessageRequester(
  "Hello", ; title
  "World", ; text
#PB_MessageRequester_Ok) ; flag
```

PB SC TU

pbcompiler.exe

. Lexical Analyzer
. Syntax Analyzer
. Code Generator

```
; heavily truncated
format MS COFF
extrn _PB_MessageRequester2@12
public PureBasicStart
section '.code' code readable executable align
4096
PureBasicStart:
 PUSH dword 0
 MOV eax,_S2
 PUSH eax
 MOV eax,_S1
 PUSH eax
 CALL _PB_MessageRequester2@12
section '.data' data readable writeable
_PB_DataSection:
public _SYS_StaticStringStart
_SYS_StaticStringStart:
_S1: dw 72,101,108,108,111,0
_S2: dw 87,111,114,108,100,0
pb_public PB_NullString
 dw 0
public _SYS_StaticStringEnd
_SYS_StaticStringEnd:
```

PureBasic.asm

FAsm

PureBasic.obj

porc
polink

PureBasic.exe

TREND MICRO

PureBasic – Linker Example (\Compilers\polink.exe)

pbcompiler.exe \Compilers**polink.exe**"
/FORCE:MULTIPLE
/OUT:"hw.exe"
/ENTRY:PureBasicStart
/SUBSYSTEM:Windows
/NODEFAULTLIB
/LIBPATH:"\Compilers"

/LIBPATH:"\PureLibraries\Windows\Libraries" PureBasic.obj SystemBase.lib StringUtility.lib UnicodeFunctions.lib MSVCRT.lib KERNEL32.lib USER32.lib GDI32.lib COMDLG32.lib ADVAPI32.lib COMCTL32.lib OLEAUT32.lib String.lib Requester.lib FileSystem.lib Date.lib Memory.lib LinkedList.lib "\compilers/objectmanager.lib" SimpleList.lib "\compilers/stringmanager.lib" OLE32.LIB Shell32.LIB Shlwapi.LIB Ole32.LIB PureBasic.res

PureBasic – Libraries

- PB libraries that PB ship with are stored in a proprietary file format
  - Linux/MacOS are stored encrypted (to be addressed later)
- Different types of libraries
  - \Compilers -> (12) Debugger.lib, libmariadb.lib, ObjectManager.lib, ObjectManagerThread.lib, Scintilla.lib, StringManager.lib, StringManagerPurifier.lib, …
  - \PureLibraries -> (116) 2DDrawing, Array, AudioCD, Billboard, Camera, CGI, Cipher, ClipboardImage, Console, Database, DatabaseMySQL, Date, DebuggerFunctions, Event, File, FileSystem, Font, Ftp, GadgetOpenGL, Help, Http, Image, ImagePlugin, LinkedList, Mail, Network, Node, Screen, SerialPort,…
  - \PureLibraries\Windows\Libraries -> (108) aclui.lib, activeds.lib, advapi32.lib, atl.lib, bdnapi.lib, cap.lib, comctl32.lib, comdlg32.lib,…
  - \PureLibraries\UserLibraries -> *this is where you store your own developed libraries.*
  - Subsystems: to change underlying libraries for specific commands. Compile time option.
    - \SubSystems\OpenGL\PureLibraries\ -> (2) Screen, Sprite
    - \SubSystems\DirectX11\PureLibraries\ -> (2) Screen, Sprite
    - Linux: gtk2 and qt, for a number of libraries.
  - \Residents -> (6) Expat.res, OpenGL.res, PureBasic.res, Scintilla.res, Unicode.res, \Unicode\Unicode.res
    - \Precompiled binary files, loaded when the compiler starts. Contains predefined structures, interfaces, macros and constants.
    - You can create your own (pbcompiler.exe /CREATERESIDENT), and .res files can be inspected with the built-in Structure Viewer GUI tool.

PureBasic – Libraries Extraction

- When compiling a PB program, the compiler dynamically extracts all relevant libraries, in their original format and save them in the current user temporary directory under the folder name “PureBasic<*GetTickCount()*>”.
  - %Temp%\PureBasic<GetTickCount()>\<library_name>.lib
  - Additionally, it creates the object file “PureBasic.obj”, the generated assembly file “PureBasic”, “Manifest”, “PureBasic.rc”, and “PureBasic.res” files in the same directory.
  - This folder is transient and gets deleted immediately right after compilation is complete
    - Figured it out using Procmon utility from Sysinternals
  - To keep the folder, you can patch the compiler executable “pbcompiler.exe” such that it doesn’t get deleted after creation.
    - Obviously, this is not a sustainable approach, and would require ‘triggering’ all the libraries (by using them in the code) so that they get all extracted. Moreover, you’d have to patch “pbcompiler.exe” for every release and on all platforms.
    - Thus, why I automated this whole process.

Dissecting PB Libraries Proprietary File Format (PFF)

PB – Libraries Proprietary File Format (PFF) – v5

```
        0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
0000h: 45 52 55 50 4C 00 00 00 34 42 49 4C 01 00 00 00  ERUPL...4BIL....
0010h: E8 45 00 00 41 6C 70 68 61 49 6D 61 67 65 00 01  èE..AlphaImage..
0020h: 00 02 49 6D 61 67 65 00 01 49 6D 61 67 65 00     ..Image..Image.
[... list of functions declaration ...]
```

{AlphaImage}

1
```
struct pb_lib
{
  std::uint8_t magic_val[5] = { '\0' }; // "ERUP" (magic value) LE!
  std::uint32_t idx_lib_data;           // starting offset of the library data starting from index 0x0C LE
  std::uint8_t lib_ver[5] = { '\0' };   // specific PureBasic library version constants (BIL3 | BIL4) LE
  std::uint32_t is_zlib_comp;           // x00 (not compressed) | x01 (compressed) LE
  std::uint32_t lib_size_decomp;        // size of lib file decompressed LE
  std::string orig_file_name;           // if a debugger library "debuggerfunctions"
} pb_lib_hdr;
```

2
```
struct pb_lib_type
{
  std::uint8_t lib_type;            // fixed x01
  std::uint8_t nb_of_ref_sys_lib;

  std::uint8_t lib_type_const;      // fixed x02; this byte is present only in case nb_of_ref_sys_lib = 0x00
                                    // thus dealing with pb_type_one_lib_hdr structure

  // shadow members                 // meaning they are not part of the actual structure, and meant only to be
                                    // used as 'helper' variables when parsing the file.
  bool is_lib_type_one = false;
  bool is_lib_type_two = false;
} pb_lib_type;
```

```
// for any value other than x00, the value
represents number of referenced pb system
libraries to follow

if (nb_of_ref_sys_lib == 0x00)
    lib_type_const == 0x02 (set)
    use pb_type_one_lib_hdr
else
    lib_type_const (is not set)
    use pb_type_two_lib_hdr
```

3
```
// this is for lib of type |01 00 02|
struct pb_type_one_lib_hdr
{
  std::string class_name;           // null terminated
  std::uint8_t nb_of_ref_intl_lib;
  std::vector<std::string> ref_intl_lib_list;
} pb_type_one_lib_hdr;
```

4
```
// this is for lib of type |01 > 0|
struct pb_type_two_lib_hdr
{
  std::vector<std::string> ref_sys_lib_list;
  std::uint8_t lib_type;            // fixed x02
  std::string lib_type_name;        // null terminated
  std::uint8_t nb_of_ref_intl_lib;
  std::vector<std::string> ref_intl_lib_list;
} pb_type_two_lib_hdr;
```

PB – Libraries Proprietary File Format (PFF) – v5

- Linux and MacOS libraries do not include `pb_lib_hdr.lib_size_decomp` data member.
- Moreover, the member `pb_lib_hdr.is_zlib_comp` is always set to zero, since the actual library payload is not compressed, but rather obfuscated with a simple algorithm. They instead have the:

```
reverse(pb_lib_hdr.lib_size_decomp) = pb_lib_hdr.lib_ver
```

  - This is how I check if I'm parsing a MacOS/Linux Or a Windows library.
  - Check slides 16/17/18 for differences/similarities between version 4 and 5 of the compiler libraries

PB – Libraries PFF Function Declaration & Library Payload – v5

```
        0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
0020h:                                             45                E
0030h: 6E 64 41 6C 70 68 61 49 6D 61 67 65 00 00 00 04  ndAlphaImage....
0040h: 00 01 00 49 6E 69 74 41 6C 70 68 61 49 6D 61 67  ...InitAlphaImag
0050h: 65 00 00 00 02 00 01 00 45 52 55 50 8F 19 00 00  e.......ERUP....
0060h: 31 54 41 44 78 9C ED 5B 0F 54 54 D7 99 7F C3 0C  1TAD.....[.TT.....
       [... library payload ...]
```

```
// shadow member
bool is_func_decl_exist = false;

struct pb_func_decl_doc
{
  std::string api_name = {};            // null terminated
  std::uint8_t nb_of_args;              // number of arguments the f. takes
  std::vector<std::uint8_t> args_list;  // the size of each argument type in bytes
  std::uint32_t api_type;               // ex., |08 00 00 00| (original f.); LE!
  std::string api_help = {};            // f. documentation; null terminated
};

std::vector<pb_func_decl_doc> pb_func_decl_vec = {};
```

```
// this struct should start at [idx_lib_data], starting from offset 0x0C
struct pb_lib_data
{
  std::uint8_t magic_val[5] = { '\0' };  // magic header value "ERUP"
  std::uint32_t length;                  // size of library file LE
  std::uint8_t type[5] = { '\0' };       // magic value "DAT1" | "DBG1"
  std::string payload = {};              // lib_payload(lib_data_length);

  // shadow member
  bool is_debug_build = false
} pb_lib_data;
```

- For MacOS/Linux, the library payload is not compressed but rather encrypted, and can be decrypted with the following algorithm:

```
key = 0x000C3500;
do
{
  dw_data = *data;
  ++data;
  decrypted = ~dw_data - key;
  key += file_size;
  *(data - 1) = decrypted;
} while (var_a + file_size > (unsigned int)data);
```

PB – Differences/Similarities Between (v4) and (v5) - Windows

- After acquiring version 4 of the compiler, I've noticed some differences and similarities at the structures levels among each of the platforms, and between the versions (v4 vs v5).

Windows	Version 4	Version 5
pb_lib_hdr.magic_val	"ERUP" (LE)	"ERUP" (LE)
pb_lib_hdr.lib_ver	"3BIL" (LE)	"4BIL" (LE)
pb_lib_hdr.lib_size_decomp	pb_lib_hdr.lib_ver in (BE)/"LIB3" *Note: size of library (dword value) decompressed is stored in the header of the compressed JCALG1 library payload, at offset 0x02.*	Decompressed library size value (LE)
Compression Algorithm	JCALG1	Zlib
pb_lib_data.payload	Encrypted and compressed *(same encryption algorithm)*	Just compressed

PB – Differences/Similarities Between (v4) and (v5) - Linux

Linux	Version 4	Version 5
pb_lib_hdr.magic_val	“ERUP” (LE)	“ERUP” (LE)
pb_lib_hdr.lib_ver	“3BIL” (LE)	“3BIL” (LE)
pb_lib_hdr.lib_size_decomp	pb_lib_hdr.lib_ver in (BE)/”LIB3”	pb_lib_hdr.lib_ver in (BE)/”LIB3”
Compression Algorithm	None of the libraries are compressed	None of the libraries are compressed
pb_lib_data.payload	Encrypted only *(same encryption algorithm)*	Encrypted only *(same encryption algorithm)*

PB – Differences/Similarities Between (v4) and (v5) - MacOS

MacOS	Version 4	Version 5
pb_lib_hdr.magic_val	“PURE” (BE)	“ERUP” (LE)
pb_lib_hdr.lib_ver	“LIB3” (BE)	“3BIL” (LE)
pb_lib_hdr.lib_size_decomp	pb_lib_hdr.lib_ver in (BE)/“LIB3”	pb_lib_hdr.lib_ver in (BE)/”LIB3”
Compression Algorithm	None of the libraries are compressed	None of the libraries are compressed
pb_lib_data.payload	Encrypted only *(same encryption algorithm)*	Encrypted only *(same encryption algorithm)*

PB – Libraries PFF – **Res**ident File Structure

```
struct res_file_header
{
  uint32_t magic;             // magic header value "ERUP" LE!
  uint32_t unknown;           // always zero
  uint32_t ResLibVersion;     // RES1-RES8 LE

  uint32_t ver_spec;          // "SRCT" (Structures) LE!
  uint32_t srct_payload_size; // LE
  std::vector<uint8_t> srct_payload(srct_payload_size);

  uint32_t cnst_marker;       // fixed "TSNC" (constants) LE!
  uint32_t tsnc_payload_size; // LE
  std::vector<uint8_t> tsnc_payload(tsnc_payload_size);

  uint32_t macr_marker;       // fixed "MACR" (macros) LE!
  uint32_t macr_payload_size; // LE
  std::vector<uint8_t> macr_payload(macr_payload_size);

  uint32_t prot_marker;       // fixed "PROT" (Interfaces) LE!
  uint32_t prot_payload_size; // LE
  std::vector<uint8_t> prot_payload(prot_payload_size);
};
```

The parser "PuBaLP" does not support the parsing of .res files.

TREND MICRO

PB – How to RE PFF Without RE'ing?

- Familiarize yourself with the framework you're researching
  - Read documentation, examples, write and test, watch behavior, take notes…
    - Google when in doubt!
  - Poke around the directories, be curious and try to open every executable and file
- The most important tool is a Hex Editor
  - Experience (& general knowledge) plays a 'major' role in identifying patterns & specific constructs
  - RE'ing binary formats is different from reversing text based formats
    - For binary formats, you need to take into account type size, whenever you're trying to make sense of a given blob of data. Look for 1-byte, word and dword type sizes. For strings, look for the null terminator character '\x00'.
    - Look for delimiters that repeats at specific offsets.
    - Look for strings that stands out, and give telltale signs about the nature of the data that follows
  - You need to look at the entire picture
    - Ex., don't just focus at the first 1-4 bytes, look at what comes next
    - Look at multiple files of the same format, and note any differences and patterns that emerge

Demo - PuBaLP - Parser

- Written in C++
- Parses PB Library files for Windows, MacOS and Linux
- Prints all the headers structures in a contextual format
- Extracts the original library file decompressed to disk, or decrypted in case of MacOS/Linux
  - Auto detects whether the parsed library file is Linux/MacOS or Windows
- Prints function declaration to the console
  - Saves all functions declarations to a file on disk as an XML file

Demo - PuBaHelper – IDA Pro Plugin

- Written in C++
- Targets only Windows PB executables
- Invoked via a popup menu from the IDA View window
- Identifies if a file is a PB one or not using three function prologues
  - Checks for the opcodes of the assembly instructions
- Asks the user to auto-apply PB IDA FLIRT signatures: Base Pure Libraries, Compiler Libraries, SubSystems DirectX11 Pure Libraries, SubSystems OpenGL Pure Libraries
  - It auto detects if the binary is 32 or 64 bit and apply respective signatures accordingly
- Provides the capability to lookup a given PB API documentation either online or in a local CHM file.

Demo - PuBaHelper – IDA Pro Plugin – Patterns: A

```
// 32_bit
const char pb_prolog_exe_a[] =
{
        "68 ? ? 00 00 "          // 68 0C 00 00 00  push    0Ch             ; Size
        "68 00 00 00 00 "        // 68 00 00 00 00  push    0               ; Val
        "68 ? ? ? 00 "           // 68 94 43 40 00  push    offset hHeap    ; Dst
        "E8 ? ? ? 00 "           // E8 EC 0F 00 00  call    memset
        "83 C4 0C "              // 83 C4 0C        add     esp, 0Ch
        "68 00 00 00 00 "        // 68 00 00 00 00  push    0               ; lpModuleName
        "E8 ? ? ? 00 "           // E8 E5 0F 00 00  call    GetModuleHandleW
        "A3 ? ? ? 00 "           // A3 98 43 40 00  mov     hmodule, eax
        "68 00 00 00 00 "        // 68 00 00 00 00  push    0               ; dwMaximumSize
        "68 00 10 00 00 "        // 68 00 10 00 00  push    1000h           ; dwInitialSize
        "68 00 00 00 00 "        // 68 00 00 00 00  push    0               ; flOptions
        "E8 ? ? ? 00 "           // E8 D2 0F 00 00  call    HeapCreate
        "A3"                     // A3 94 43 40 00  mov     hHeap, eax
};
```

Demo - PuBaHelper – IDA Pro Plugin – Patterns: B

```
// 32_bit
const char pb_prolog_dll_a[] =
{
        "83 7C 24 08 01 "        // 83 7C 24 08 01     cmp     [esp+fdwReason], 1
        "75 ? "                  // 75 19              jnz     short check_reason_2
        "8B 44 24 04 "           // 8B 44 24 04        mov     eax, [esp+hinstDLL]
        "A3 ? ? ? 10 "           // A3 40 43 00 10     mov     dword_hinstdll, eax
        "E8 ? ? ? 00 "           // E8 4E 00 00 00     call    heap_create
        "FF ? ? ? ? ? "          // FF 35 40 43 00 10  push    dword_hinstdll
        "E8 ? ? ? 00 "           // E8 88 00 00 00     call    xor_eax
        "83 7C 24 08 02 "        // 83 7C 24 08 02     cmp     [esp+fdwReason], 2
        "75 ? "                  // 75 0B              jnz     short check_reason_0
};
```

TREND MICRO

Demo - PuBaHelper – IDA Pro Plugin – Patterns: C

```
// 64_bit. This hits on both the EXE as well as the DLL versions
const char pb_prolog_exe_b[] =
{
        "48 ? ? ? "                 // 48 83 EC 28                  sub     rsp, 28h
        "49 ? ? ? 00 00 00 "        // 49 C7 C0 30 00 00 00         mov     r8, 30h         ; Size
        "48 ? ? "                   // 48 31 D2                    xor     rdx, rdx        ; Val
        "48 ? ? ? ? ? ? 00 00 00 "  // 48 B9 50 24 01 40 01 00 00 00 mov     rcx, offset hHeap ; Dst
        "E8 ? ? 00 00 "             // E8 E3 0F 00 00               call    memset
        "48 ? ? "                   // 48 31 C9                     xor     rcx, rcx        ; lpModuleName
        "E8 ? ? ? 00 "              // E8 E1 0F 00 00               call    GetModuleHandleW
        "48 ? ? ? ? ? ? "           // 48 89 05 2C 14 01 00         mov     cs:hInstance, rax
        "4D ? ? "                   // 4D 31 C0                     xor     r8, r8          ; dwMaximumSize
        "48 ? ? 00 10 00 00 "       // 48 C7 C2 00 10 00 00         mov     rdx, 1000h      ; dwInitialSize
        "48 ? ? "                   // 48 31 C9                     xor     rcx, rcx        ; flOptions
        "E8 ? ? ? 00 "              // E8 CE 0F 00 00               call    HeapCreate
        "48 "                       // 48 89 05 0B 14 01 00         mov     cs:hHeap, rax
};
```

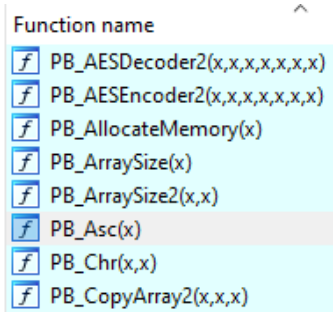
Generating IDA FLIRT Signatures

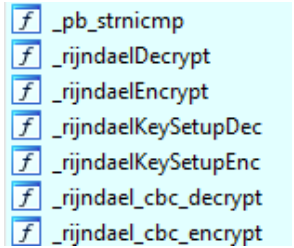
- Get the SDK “flair” file (you need a paid version of IDA Pro)
- Instructions are Windows specific, but all flair tools are available for Linux and MacOS to generate flirt signatures
- Once all PB libraries are extracted, you’re ready to generate the signature files
- Use pcf.exe (COFF parser) to generate library pattern file (.pat)
- Use sigmake.exe (Signature file maker) to generate the final signature file (.sig)
  - Fix collisions, if any, and attempt to rebuild the signature file
  - Most of the collisions are related to debug functions

- The plugin comes with four major signature files, targeting the latest version of PB (5.71), 32 and 64 bit builds
- You can either use the plugin to auto apply the signatures, or apply them at your discretion

Case Study – PureLocker

- This ransomware was first documented by Intezer, on Nov 12, 2019
- It is written in the PB PL
  - It is unknown which version of PB compiler was used
  - Uses custom function, anti-analysis, PB based encryption functions…
  - Originally, IDA identifies only 18/231 functions
  - After applying PB signatures, IDA recognizes **124**/237, and discovers new functions (6)
    - More importantly, is the recognition of the cryptographic calls





Case Study – Xml.pb

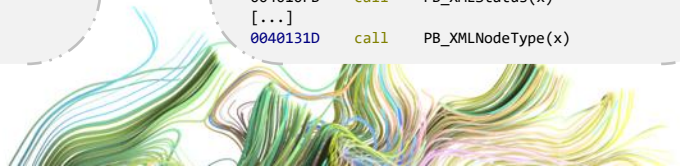
- This example was taken from the \Examples\Sources PB installation folder.
- It is responsible for loading an XML file, parsing it and displaying it
- After compiling it with PB compiler v5.71
  - Originally, IDA identifies only 41/559 functions
  - After applying PB signatures, IDA recognizes **300**/585, and discovers new functions (24)
    - The rest of the unidentified functions are either too small or unimportant to the core functionality of the tool

original

```
00401094    call    sub_40230F
00401099    mov     edx, dword_427584
[...]
004010B7    push    eax
004010B8    call    sub_4032FA
004010BD    push    offset lpFileName
004010C2    call    sub_40C340
004010C7    push    lpFileName
004010CD    mov     edx, offset word_426024
004010D2    pop     ecx
004010D3    call    sub_402040
[...]
004010E9    call    sub_4025B5
004010EE    and     eax, eax
004010F0    jz      loc_4012A7
004010F6    push    0
004010FB    call    sub_4025C7
[...]
0040131D    call    sub_402687
```

flirted

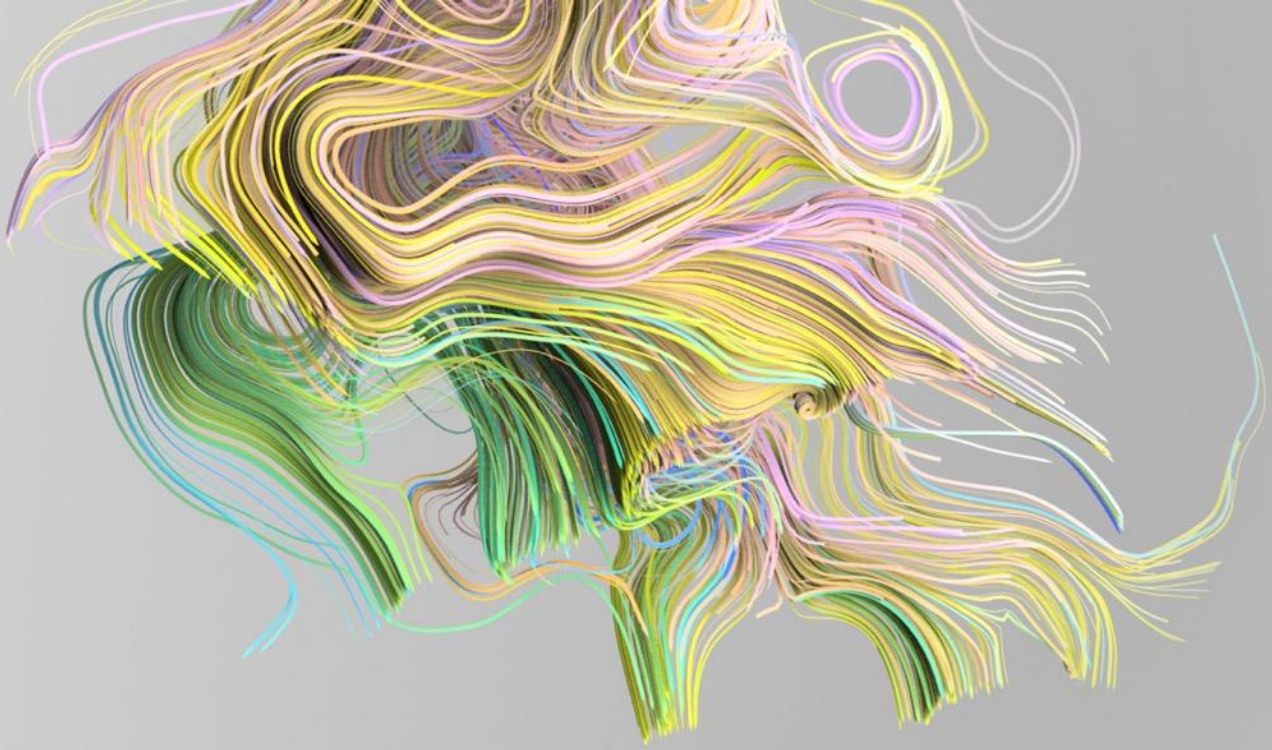
```
00401094    call    PB_InitXML()
00401099    mov     edx, dword_427584
[ ... ]
004010B7    push    eax
004010B8    call    PB_OpenFileRequester(x,x,x,x,x)
004010BD    push    offset lpFileName
004010C2    call    SYS_AllocateString4(x,x)
004010C7    push    lpFileName
004010CD    mov     edx, offset word_426024
004010D2    pop     ecx
004010D3    call    _SYS_StringEqual
[...]
004010E9    call    PB_LoadXML(x,x)
004010EE    and     eax, eax
004010F0    jz      loc_4012A7
004010F6    push    0
004010FB    call    PB_XMLStatus(x)
[...]
0040131D    call    PB_XMLNodeType(x)
```



Challenges, Recommendations and Future Work

- The biggest challenge is collecting old versions of PB compiler, for all platforms, so that a KB of IDA FLIRT signatures is created for at least all major versions/releases.
- Provide FLIRT signatures for the Linux/MacOS versions of the compiler
- Update parser PuBaLP to parse .res files
- Update IDA PuBaHelper plugin
  - To load XML file (function declaration) and populate identified PB functions with relevant metadata
  - To detect MacOS and Linux PB binaries
  - Manually cover some unique unidentified PB functions in a standalone signature file (we could simply use ida2pat plugin from FE)

- When parsing such structures, make sure you don't use 'global offsets' to track where in the file/memory the next structure starts or ends.
  - It is better if you copy each structure's block of data separately, and parse it independently, whenever possible



Conclusion

Thank You

Q & A