REVERSING LABS

Mario Vuksan & Tomislav Pericin

BlackHat USA 2012, Las Vegas

# File Disinfection Framework: Striking back at polymorphic viruses

## AGENDA

**Introduction to…**

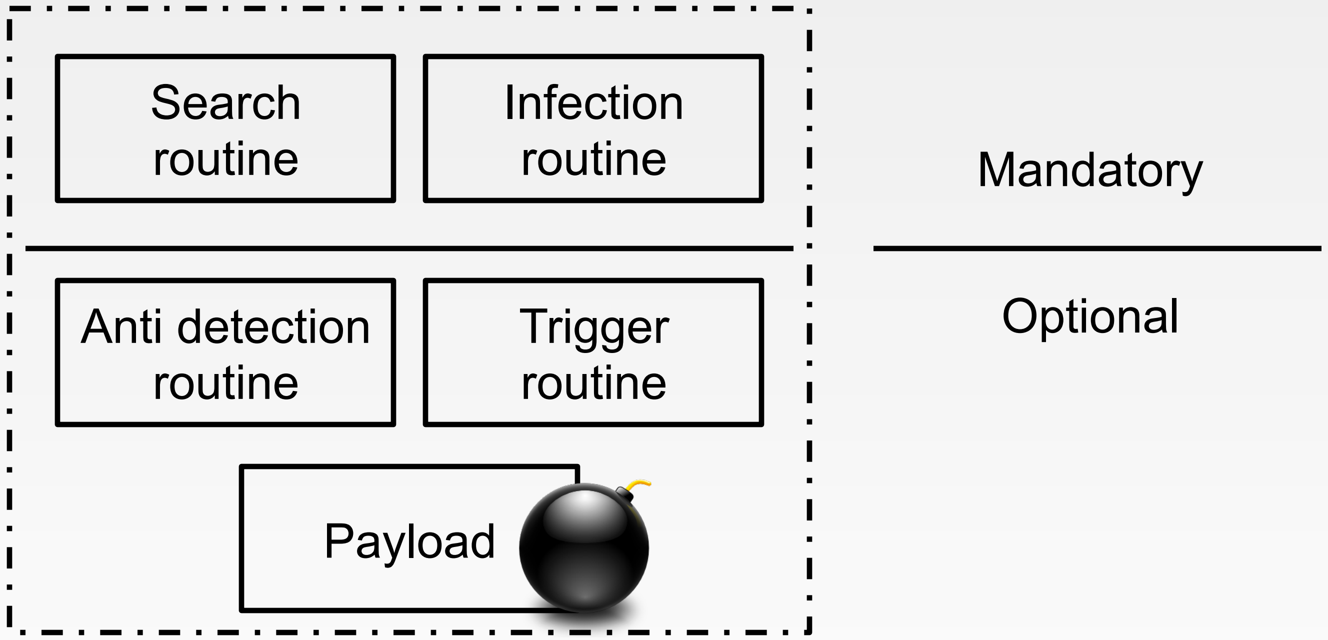
- Computer viruses
- Polymorphic virus problem
- File disinfection framework

**File disinfection framework**

- Static file disinfection
  - Creating a simple static disinfector
- Emulator aided file disinfection
  - Emulator design
  - Creating a simple emulator aided disinfector

## WHAT IS A COMPUTER VIRUS?

“Computer virus is a computer program that can replicate itself and spread from one computer to another.”



## HISTORY OF COMPUTER VIRUSES

**1949**

- The first academic work on the theory of computer viruses was done in 1949 by John von Neumann

**1971**

- The Creeper virus, an experimental self-replicating program, is written by Bob Thomas at BBN Technologies. Creeper infected DEC PDP-10 computers running the TENEX operating system. The Reaper program was later created to delete Creeper.

## HISTORY OF COMPUTER VIRUSES

**1984**

- The term 'virus' is coined by Frederick Cohen in describing self-replicating computer programs. In 1984 Cohen uses the phrase "computer virus" to describe the operation of such programs in terms of "infection". He defines a 'virus' as "a program that can 'infect' other programs by modifying them to include a possibly evolved copy of itself."

**1987**

- Appearance of the Vienna virus, which was subsequently neutralized—the first time this had happened on the IBM platform.
- Appearance of Lehigh virus, boot sector viruses such as Yale from USA, Stoned from New Zealand, Ping Pong from Italy, and appearance of first self-encrypting file virus, Cascade.

## HISTORY OF COMPUTER VIRUSES

**1990**

- Mark Washburn working on an analysis of the Vienna and Cascade viruses with Ralf Burger develops the first family of polymorphic virus: the Chameleon family.

**1998**

- The first version of the CIH virus appears.

**2001 – 2002**

- Zmist (also known as Zombie.Mistfall) is a metamorphic computer virus created by the Russian virus writer known as Zombie.
- Simile (computer virus) is a metamorphic computer virus written in assembly.

## FILE INFECTION PROBLEM

Polymorphic viruses

- File infection is the ultimate polymorphism
- File infector replicates self into a host of unique containers
- One strain can generate 100K+ of unique infected samples
- Response is difficult
    - Focus on reducing polymorphisms to a single infection
    - Classifying multi-infections
    - Disinfection is like surgery
        - Remove offending payload surgically
        - Make sure there’s no remaining malignant content
        - Close the incision areas so patient can walk again
    - Failure is severe
        - Potential for re-infection
        - Potential for system downtime and loss of data

## DISINFECTION DILEMMA

Disinfect or re-image the machine?

- Consumer
    - Always disinfect, rarely re-image
    - Most users do not back up frequently or at all
    - Loss of personal documents and media trumps security risks
    - Is served by AV product
- Enterprise
    - Always re-image if possible
    - But re-image is not always practical
    - IT staff will quit if we re-imaging over 10K machines is requested
    - Disinfection to be followed by staged re-imaging is preferrable
    - Yet, who will write custom disinfectors for the Enterprise?

## WHY DID WE CREATE FDF?

What's its purpose and who will benefit from it?

- Sponsored by DARPA CFT program
- Purpose
  - Speed up development of disinfection routines and utilities
  - Increase the quality
  - Reduce the risk of failure
  - Collaborate with the community
    - Open source modules
- Benefactors
  - AV companies for mass disinfection modules
  - CERT teams for specialized projects
  - ISPs for Botnet infection reduction
  - Enterprises for custom disinfectors

## FILE DISINFECTION FRAMEWORK

## TITANENGINE

**Open source library for PE file processing**

- Version 1.0
  - Historic version, purely dynamic file processing centered
- Version 2.0
  - Presented at BlackHat USA 2009
  - Total rewrite from ASM to C
  - Many improvements in the field of dynamic file processing
- Version 3.0
  - Presented at Hack In the Box 2012
  - Total rewrite to C++
  - Purely static file processing centered
- Version 3.1
  - Presenting at BlackHat 2012
  - Inclusion of file disinfection components
  - Static file processing enriched with the x86 emulator

# FILE DISINFECTION FRAMEWORK

**Open source library for PE file disinfection**

- Version 3.1 (Windows x86-x64)
  - Static portable executable manipulation
  - Portable executable integrity validation and recovery
  - Dynamic portable executable building
  - Import recovery with hash databases
  - Dynamic decrypter building
  - Advanced x86 emulator
- Statistics:
  - Over a 100 user exposed APIs
  - Over 40,000 lines of C++ code

## USING STATIC FILE FUNCTIONS

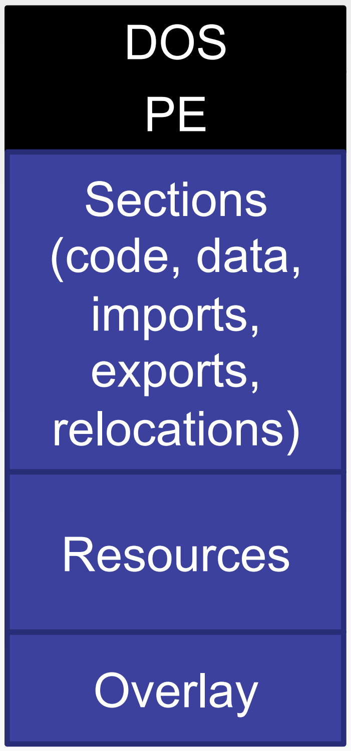
## STATIC FILE MANIPULATION

**Features**

- Static PE32/32+ file format processing functionality
    - Ability to read, modify and create new PE files
    - Ability to read, modify, split, merge and create PE sections
    - Ability to read, modify and create individual PE tables
- Support for decompressing large number of formats
- Support for building custom dynamic decrypters
- Support for import hash to original name reverting
- PE file format validation, malformation detection, damage assessment and recovery

## WORKING WITH FILES

### Loading files and working with their content

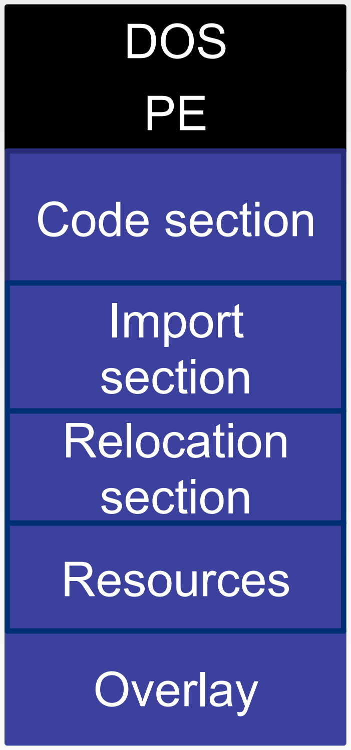


Traditional PE format layout

- TitanEngine3 file transformation process:
  1. Portable executable headers are read from the file on disk
  2. Sections are mapped into memory converting the file into a flat memory model
  3. All modifications to file are done in memory & memory is allocated only when a specific section is being modified
  4. No content is saved until explicitly instructed
  5. When changes are saved the entire file is rebuild to ensure format validity

## WORKING WITH PE TABLES

### Loading files and working with their content

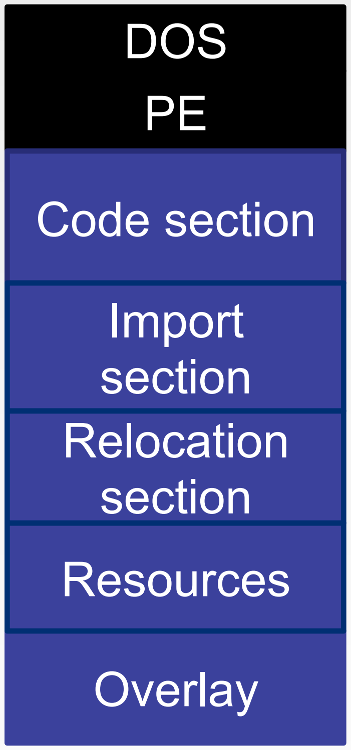


Traditional PE format layout

- TitanEngine3 can build new:
  - Import table
  - Export table
  - Relocation table
  - Resource table
  - TLS table
  - Overlay

## WORKING WITH PE TABLES

### Loading files and working with their content

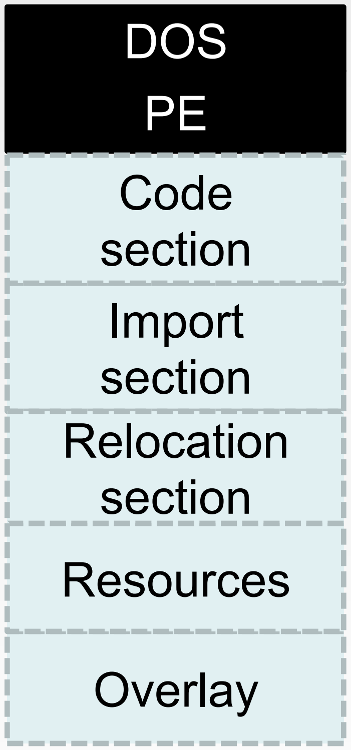


Traditional PE format layout

- TitanEngine3 can read/enumerate:
    - Bound import table
    - Delay import table
    - Exception table
    - Debug table
- Data is enumerated through:
    - msgpack serialization
    - User callbacks

# CREATING A NEW PE FILE

## Dynamic portable executable building

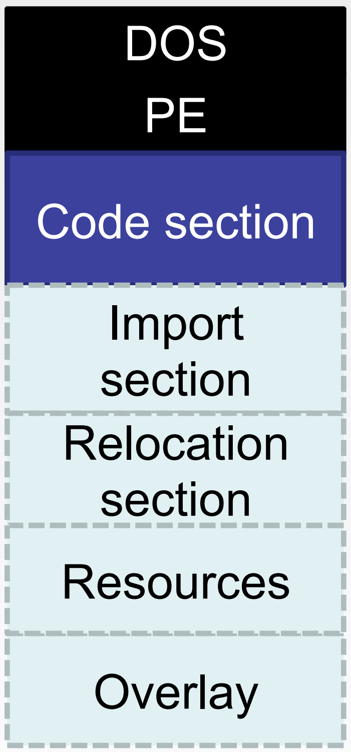


Traditional PE format layout

- Creating a new PE32/PE32+ file
    - **titan_create_file** API is used to create a new PE32/PE32+ file in memory.
    - No sections exist at this time and they must be added before storing any data at that location.
    - **titan_set_pe_header** API is used to update the PE header data.
    - Default PE header can be accessed and its parameters can be changed at any time.

# ADDING A CODE SECTION

## Dynamic portable executable building

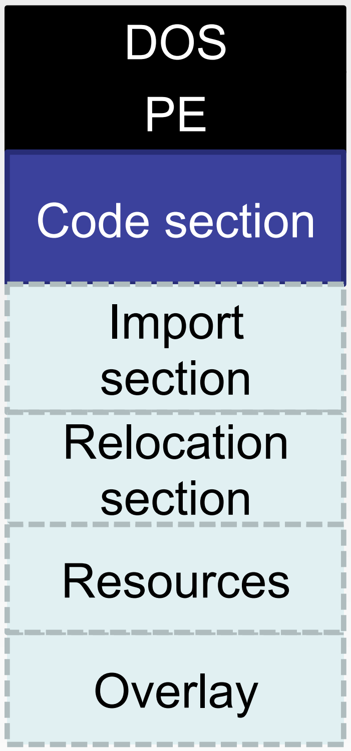


Traditional PE format layout

- Adding sections to PE32/PE32+ file
  - **titan_add_new_section** API is used to create a section inside the PE header.
  - **titan_get_content** API is used to read data from PE sections.
  - **titan_set_content** API is used to write data to PE sections.
  - Last section can always be increased by writing past its end but writing must start with the current section limits.

## WORKING WITH SECTIONS

### Dynamic portable executable building

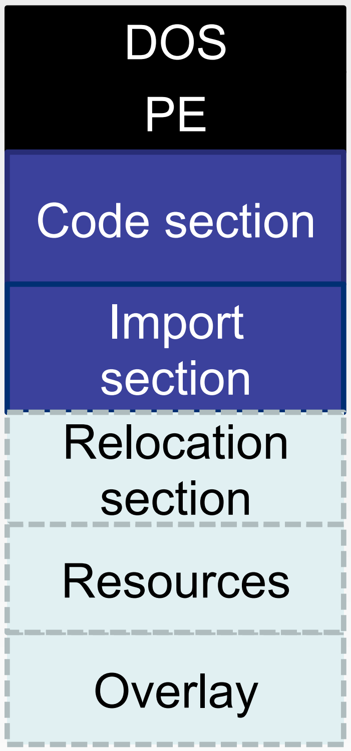


Traditional PE format layout

- Working with section content
  1. Accessing section content via relative virtual addresses. This enables the access to the **current** content of the section.
  2. Accessing section content via physical addresses. This enables the access to content as it **was** on the disk.

# ADDING AN IMPORT TABLE

## Dynamic portable executable building

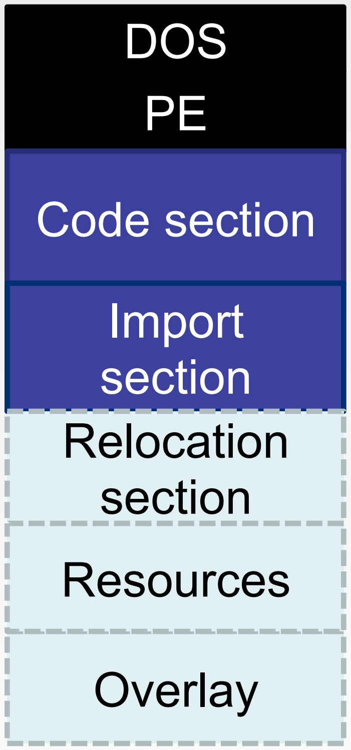


Traditional PE format layout

- Adding a new import table
  - kernel32.dll
    - GetModuleHandleA
    - LoadLibraryA
    - GetProcAddress
  - user32.dll
    - MessageBoxA

# ADDING AN IMPORT TABLE

## Dynamic portable executable building

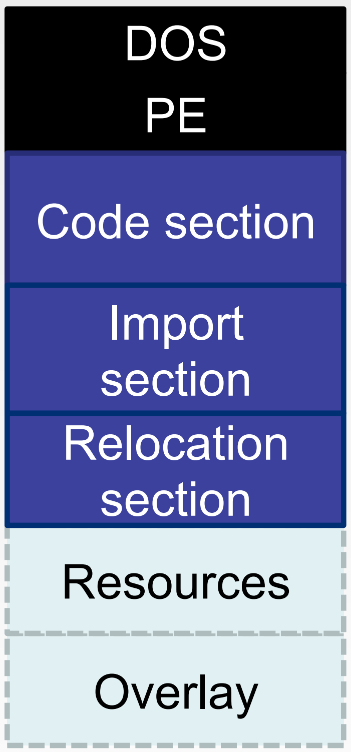


Traditional PE format layout

- Adding a new import table
  - **titan_add_import_library** API is used to add new imported DLL file.
  - **titan_add_import_function** API is used to add functions to added DLLs. This API is called after the DLL entry has been added. API pushes with this function belong to DLL which was added last.
  - **titan_write_import_table** API is used to write the import table data we pushed to the engine to the specified location.

# ADDING A RELOCATION TABLE

## Dynamic portable executable building

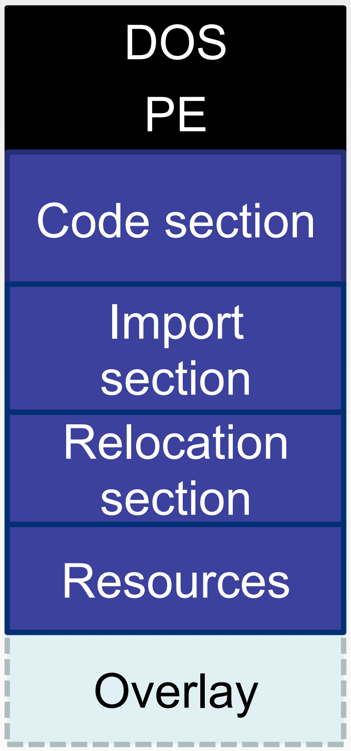


Traditional PE format layout

- Adding a new relocation table
  - **titan_add_base_relocation** API is used to add addresses from code and data section(s) which need to be relocated.
  - **titan_write_relocation_table** API is used to write the relocation table data we pushed to the engine to the specified location.

# ADDING A RESOURCE TABLE

## Dynamic portable executable building

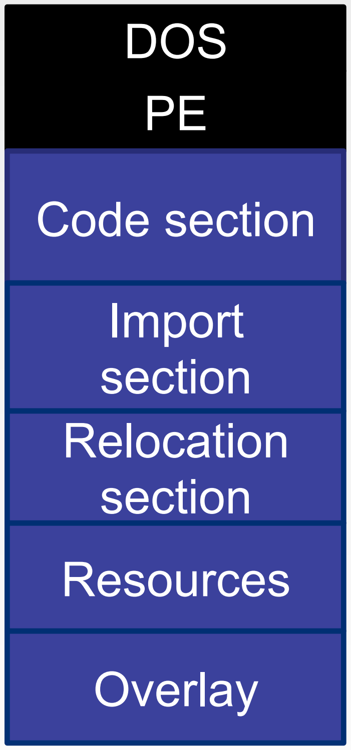


Traditional PE format layout

- Adding a new resource table
  - **titan_add_resource_data** API is used to add new resources to the file. Every resource is defined with its name, type, language, code page and data.
  - **titan_write_resource_table** API is used to write the resource table data we pushed to the engine to the specified location.

## WORKING WITH OVERLAY

### Dynamic portable executable building

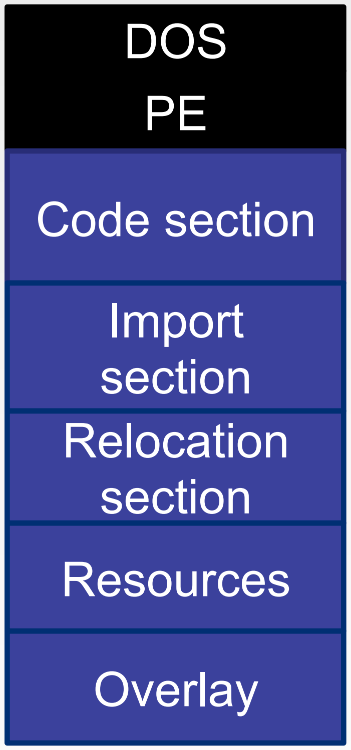


Traditional PE format layout

- Finding & reading the overlay
  - **titan_find_overlay** API is used to determine if the file has an overlay.
- Adding overlay to a file
  - **titan_add_overlay** API is used to add new overlay data to the file.
  - Data can be added to file in chunks. They are subsequently added to the end of the rebuild file.
- When rebuilding the file you can chose to preserve or remove debug table and certificate.

## SAVING CHANGES

### Dynamic portable executable building



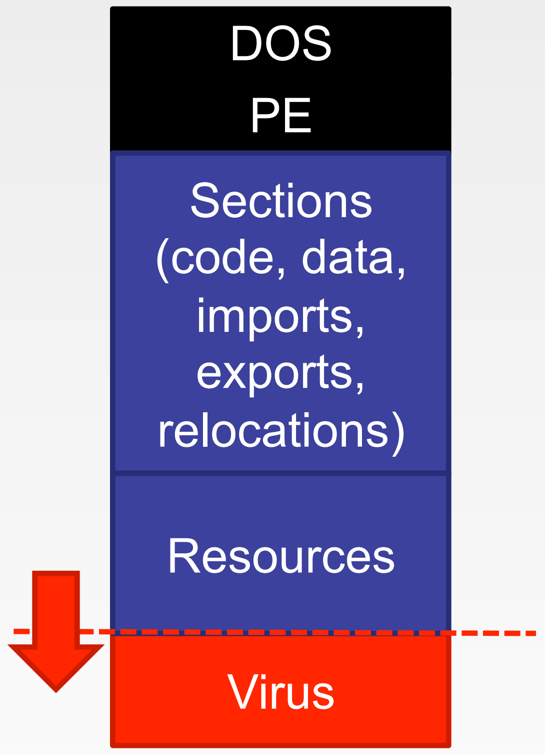
Traditional PE format layout

- Exporting work from engine to disk
  - **titan_export_file** API exports the current state of the PE header and the sections from memory to a new file on disk.
  - When exported file is reconstructed and its section content physical size is minimized so that sections with no data take-up no space on disk.

## DISINFECTING WIN32.AWFULL

## VIRUS.WIN32.AWFULL

### General virus information



Infected file layout

Malware type:

- PE32 file infector

Aliases:

- W32/Awfull.2376 (McAfee)
- W32.Nector (Symantec)
- W32/Awfull.2376 (Avira)
- W32/Awfull-B (Sophos)

Versions:

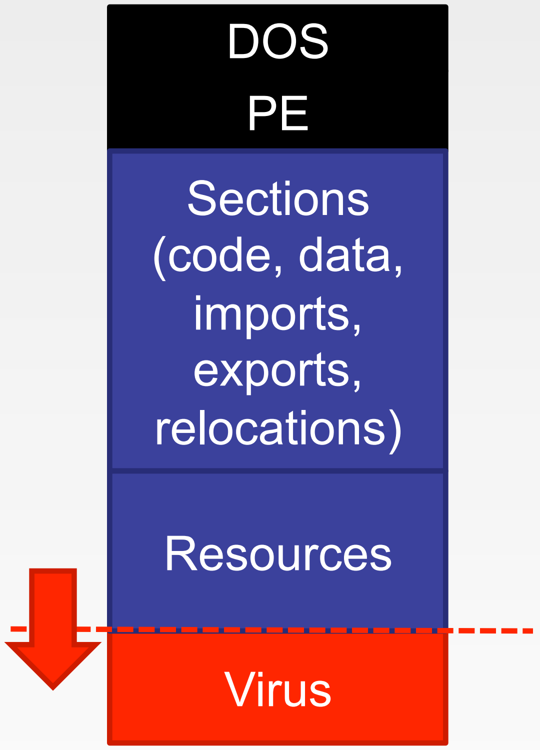
- 2376, 3254, 3318, 3574 (based on size)

Comment:

- Is actually awful

# VIRUS.WIN32.AWFULL.3318

## Infected file behavior



Infected file layout

Virus body encryption

- Original entry point is encrypted

Anti reversing protections

- IsDebuggerPresent
- CreateFileA (\\.\SICE & \\.\NTICE)

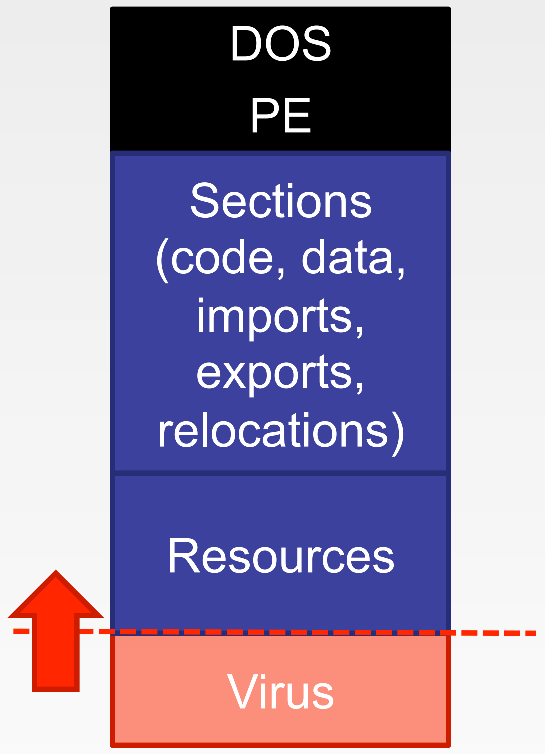
Uses VBScript to infect other .vbs files

Bugs:

- MOV ESI,[address] instead of offset
- Debugger checks don't check API return
- Can't find files to infect because of wrong parameters passed to FindFirstFileA
- Closing non open handles
- Destroys original overlay

# DISINFECTION PROCEDURE

## Creating disinfection routine with FDF



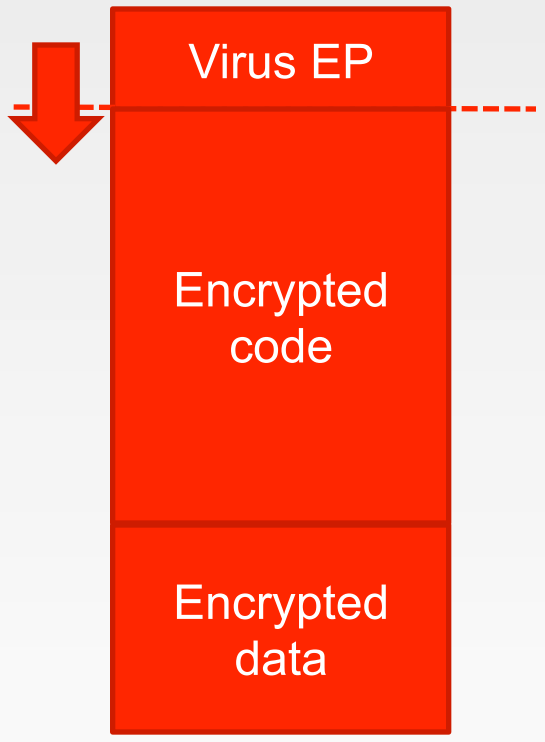
Infected file layout

Disinfection procedure steps:

1. Check if the file is infected
   - Determine virus version
   - Determine if it’s a virus body
2. Get the virus code delta (EBP)
3. Decrypt encrypted virus body
4. Read the original entry point address
5. Resize the last section (remove virus)
6. Update the PE header with new OEP
7. Save the disinfected file

# DECRYPTING THE VIRUS BODY

## Dynamically creating decryption routines



Infected file layout

Decryption procedure steps:

1. Extract the decryption algorithm from the unencrypted virus entry point
2. Apply the extracted algorithm to encrypted code & data
3. Verify that decryption has succeeded
4. Extract the needed decrypted data
   - Original Entry point address

## DISINFECTION DEMO

## CREATING DECRYPTERS

- Decrypters execute user defined x86-x64 code
- They are exclusively used to decrypt polymorphic code
- Decryption direction is either from start to end or in reverse
- Decryption is performed in steps of 1, 2, 4 or 8 bytes
- Decryption is loop based and each pass can access only one encrypted block of 1, 2, 4 or 8 bytes at the time
- Decrypters can consist of most x86-x64 instructions
- Decrypters emulate instruction code and flags
- Decryption can use callbacks and be traced with TF set
- User can terminate execution by calling HLT instruction

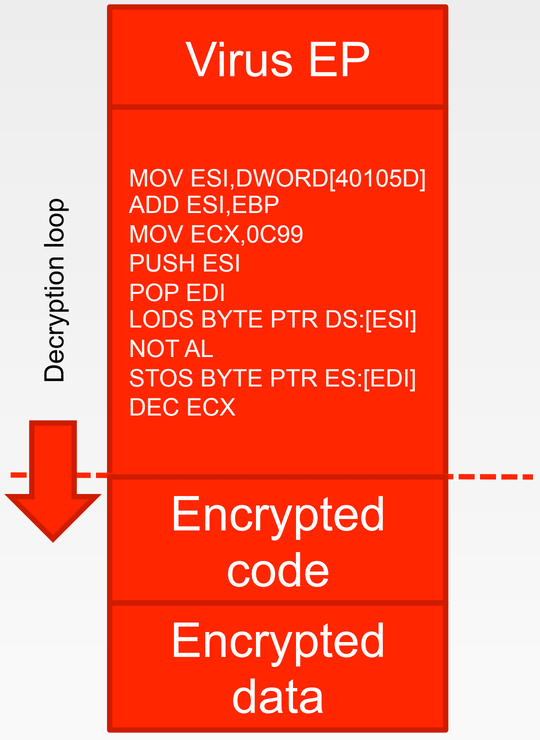
## VM INSTRUCTION FORMAT

- Decrypter instructions are stored in two streams
  - Virtual machine initialization stream (executed once)
  - Virtual machine code stream (performs decryption)
- Example: **mov al,byte ptr[current_encrypter_buffer_pos]**

```
struct code_instruction_t
{
        opcode_t        opcode;             /* op_mov_k */
        opcode_type_t   type;               /* op_t_reg_s_mem1_k */
        register_t      target;   /* reg_al_k */
        register_t      source;             /* reg_none_k */
        reg64_t  immediat;        /* 0 */
};
```

# DECRYPTING THE VIRUS BODY

## Dynamically creating decryption routine



```
MOV ESI,DWORD[40105D]
ADD ESI,EBP
MOV ECX,0C99
PUSH ESI
POP EDI
LODS BYTE PTR DS:[ESI]
NOT AL
STOS BYTE PTR ES:[EDI]
DEC ECX
```

Infected file layout

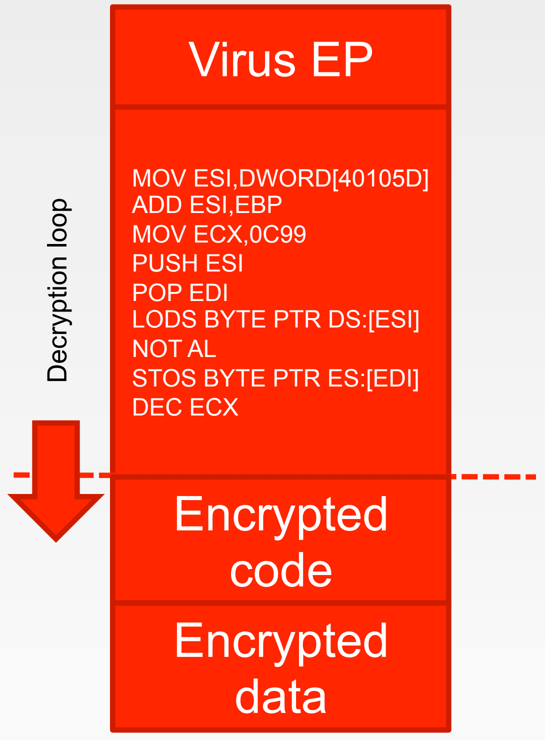
```
code_instruction_t decypter[]
{
/* MOV AL, BYTE PTR["ESI"] */
        { op_mov_k, op_t_reg_s_mem1_k, reg_al_k, reg_none_k, 0 },
/* NOT AL */
        { op_not_k, op_target_reg_k, reg_al_k, reg_none_k, 0 },
/* MOV BYTE PTR["ESI"], AL */
        { op_mov_k, op_t_mem1_s_reg_k, reg_none_k, reg_al_k, 0 }
};
```

Note: This VM code loops (ECX / 1) times

(and yes the virus should use an offset for mov esi, that's a bug in the virus code)

## DECRYPTING THE VIRUS BODY

### Dynamically creating decryption routine



```
MOV ESI,DWORD[40105D]
ADD ESI,EBP
MOV ECX,0C99
PUSH ESI
POP EDI
LODS BYTE PTR DS:[ESI]
NOT AL
STOS BYTE PTR ES:[EDI]
DEC ECX
```

Infected file layout

- Creating a dynamic decrypter
  - **titan_create_decrypter** API is used to create a new x86-x64 dynamic decrypter.
  - **titan_add_decrypter_code** API is used to add new instructions to the selected decrypter.
  - **titan_decrypt_data** API executes the virtual machine code and decrypts the data in steps of 1, 2, 4 or 8 bytes.
  - We use the ECX value to set the number of bytes to decrypt.

## VM INSTRUCTION FORMAT

**Decrypters can handle conditional jumps**

```
code_instruction_t decypter[]
{
/* MOV AL, BYTE PTR[“ESI”]
            { op_mov_k, op_t_reg_s_mem1_k, reg_al_k, reg_none_k, 0 },
/* STC */
            { op_stc_k, op_unknown_k, reg_none_k, reg_none_k, 0 },
/* JNC @4 */
            { op_jnc_k, op_unknown_k, reg_none_k, reg_none_k, 4 },
/* INC AL */
            { op_inc_k, op_target_reg_k, reg_al_k, reg_none_k, 0 },
/* 4: XOR AL, 0x90 */
            { op_xor_k, op_t_reg_s_const1_k, reg_al_k, reg_none_k, 0x91 }
};
```

## VM INSTRUCTION FORMAT

**Decrypters have the basic push/pop stack (ESP is affected)**

```
code_instruction_t decypter[]
{
/* MOV AL, BYTE PTR[“ESI”]
            { op_mov_k, op_t_reg_s_mem1_k, reg_al_k, reg_none_k, 0 },
/* PUSH EAX */
            { op_push_k, op_source_reg_k, reg_none_k, reg_eax_k, 0 },
/* POP EBX */
            { op_pop_k, op_target_reg_k, reg_ebx_k, reg_none_k, 0 },
/* INC BL */
            { op_inc_k, op_target_reg_k, reg_bl_k, reg_none_k, 0 },
/* MOV BYTE PTR[“ESI”], BL */
            { op_mov_k, op_t_mem1_s_reg_k, reg_none_k, reg_bl_k, 0 }
};
```

## VM INSTRUCTION FORMAT

**Decrypters can call user mode callbacks**

```
code_instruction_t decypter[]
{
            { op_mov_k, op_t_reg_s_mem1_k, reg_al_k, reg_none_k, 0 },
            { op_stc_k, op_unknown_k, reg_none_k, reg_none_k, 0 },
            { op_jnc_k, op_unknown_k, reg_none_k, reg_none_k, 4 },
            { op_inc_k, op_target_reg_k, reg_al_k, reg_none_k, 0 },
            { op_xor_k, op_t_reg_s_const1_k, reg_al_k, reg_none_k, 0x91 },
/* Callback can change register state with titan_set_decrypt_context */
            { op_call_k, op_unknown_k, reg_none_k, reg_none_k, 0 },
            { op_push_k, op_source_reg_k, reg_none_k, reg_eax_k, 0 },
            { op_pop_k, op_target_reg_k, reg_ebx_k, reg_none_k, 0 },
            { op_mov_k, op_t_mem1_s_reg_k, reg_none_k, reg_bl_k, 0 }
};
```

## FDF EMULATOR DESIGN

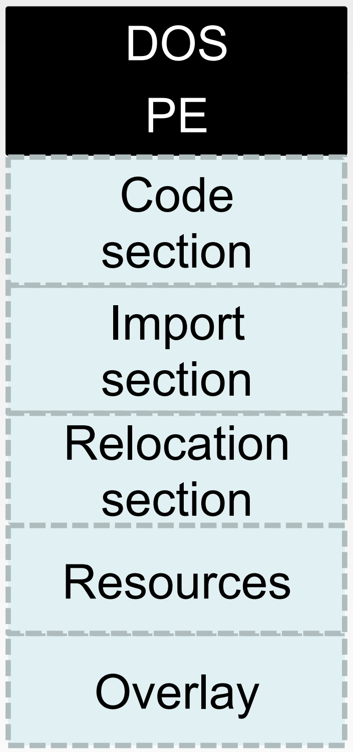
## EMULATOR DESIGN CONCEPT

**Features**

- Emulating x86 Windows environment
- Emulating PEB, TEB and SEH structures
- Emulating user mode libraries
  - Over 100 functions from kernel32.dll and user32.dll
  - Dynamically building custom libraries from hash databases
  - Loading user defined custom libraries into the process
- Emulating file system
  - Customizable drives (type, size, name, serial, …)
  - Supporting file mapping, attributes, time stamps and sharing
- Support for multiple breakpoints per address, page or API
- Support for hooking and replacing API functionality
- Support for high level code execution in the emulated process

# LOADING THE FILE

## Loading the file inside the emulated environment



PE32 file from disk (executable or dynamic link library)

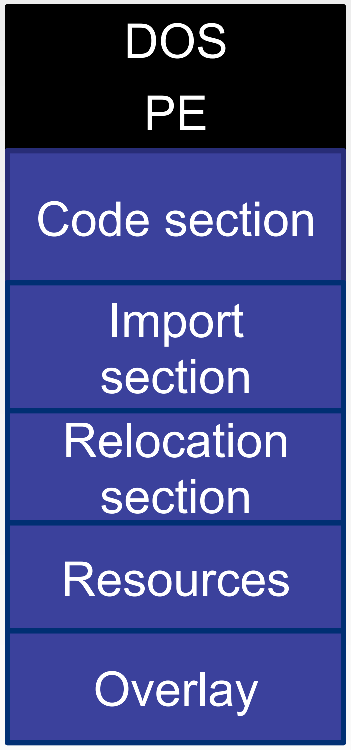
- Initializing the emulation process
  - **titan_open_file** API is used to load the file from disk.
  - While most PE files can be loaded only x86 PE32 executable and dynamic link library files can be emulated.
  - At this point all necessary static operations can be performed.

## VALIDATING & REPAIRING FILES

- During the last years BlackHat ReversingLabs has published a number of PE malformations that can be introduced to the format
- These complex malformation are now the integral part of the TitanEngine PE file validation process
- In addition to malformations files can also be damaged which is quite common for virus infections
- **titan_validate_file** API performs malformation checks and estimates the damage done to the file.
- File validation is strict and implemented to reflect the PECOFF documentation not the implementation in various version of the OS loader
- If the damaged and is estimated to be in recoverable state then you can use **titan_repair_file** API to recover the damage

# EMULATING THE PROCESS

## Creating a new process inside the emulator



PE32 file from disk (executable or dynamic link library)

- Initializing the emulation process
  - **titan_vm_create_process** API is used to initialize the virtual process environment.
  - This API performs the environment initialization process that is similar to the one which Windows performs.
  - This executes the following steps:
    - Creates a new emulated instance
    - Initializes the emulated file system
    - Maps the file into memory
    - File is relocated if necessary
    - Loads the necessary dependencies

## RUNNING THE EMULATOR

- **titan_vm_run_process** API is used to run the emulation process.
- Emulation can be controlled via the set of predefined callbacks that occur on breakpoints and other events
- List of supported Windows events:

```
enum debug_event_code_t
{
            initialize_debug_event_k,
            exception_debug_event_k,
            create_thread_debug_event_k,
            create_process_debug_event_k,
            exit_thread_debug_event_k,
            exit_process_debug_event_k,
            load_dll_debug_event_k,
            unload_dll_debug_event_k,
            output_string_debug_event_k,
            rip_event_k
};
```

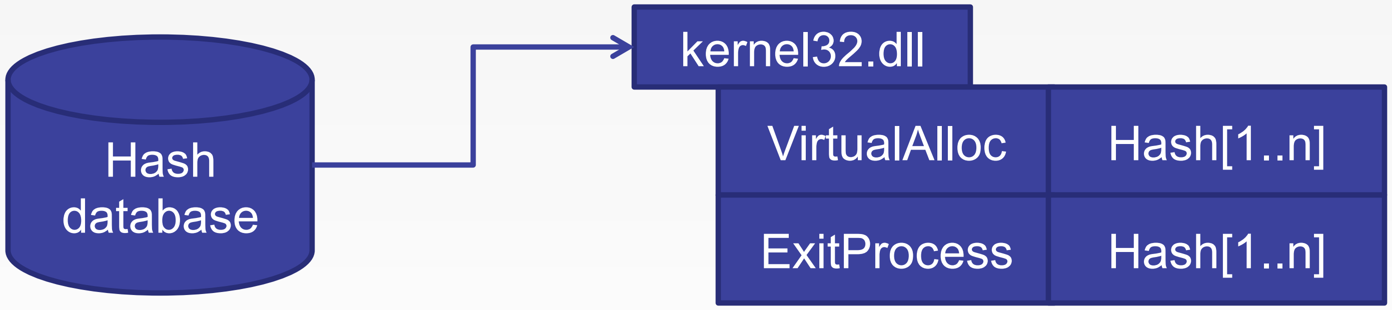
## DEPENDENCIES

### Loading application dependencies

- Dependencies can be loaded statically and dynamically
- Statically loaded dependencies
  - Can be generated be the user during process initialization
  - Can be automatically generated from hash databases
- Dynamically loaded dependencies
  - Can be loaded from the virtual file system
  - Can be generated by the user before loading
  - Can be automatically generated from hash databases

## HASH DATABASES

- Hash databases are user created files that have the list of module exported functions
- Hash databases enable reverse hash queries
- Every entry in the hash database is the name of the function with its corresponding hashes
- List of internal hashes is small and limited to algorithms commonly found in packers or viruses



## CREATING HASH DATABASES

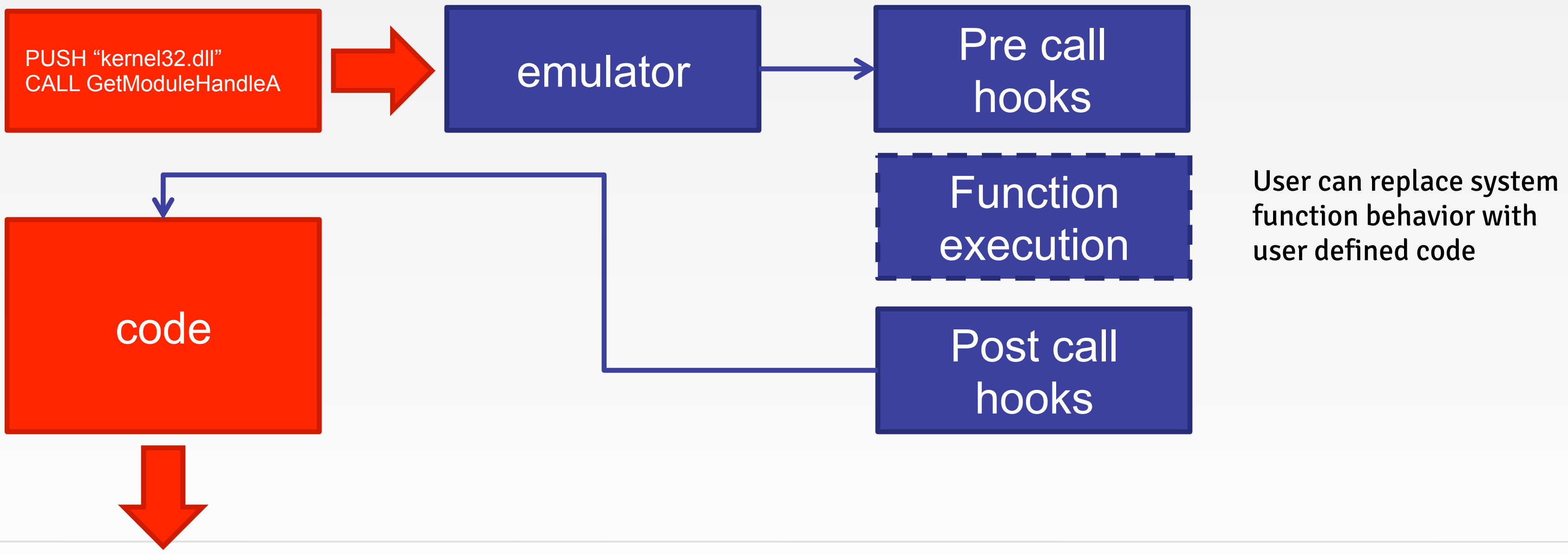
- Hash databases can automatically import PE32/32+ libraries
- Existing databases automatically upgrade when new algorithms are added inside the TitanEngine
- Expanding hash databases with new algorithms doesn’t require source code modification
  - User can add more hashes with custom “decryption” algorithms
  - They are the same as the ones discussed before but only used for hashing
  - Algorithms you write are stored in the database itself
  - Resulting RAX value stores the hash value

## USING HASH DATABASES

- Hash databases can be used for reverse hash queries
- User can push multiple hash databases to the emulator
- If multiple databases have the same library inside them only the database which was pushed later is used
- Statically imported libraries are created automatically from all available hash databases
  - If the requested library isn’t found in a hash database it is automatically created by the engine

## WORKING WITH FUNCTIONS

- Emulator has the predefined set of emulated functions
- List of supported functions can be expanded dynamically
- **titan_vm_hook_function** API can be used to both intercept emulated APIs and define the virtual machine behavior when the specified API is called



## BREAKPOINTS

- Emulator supports various breakpoint types:
  - On page
  - On address
  - On instruction (can be combined with address or page)
- Breakpoint types can be combined with triggers:
  - On write
  - On read
  - On execution
- Breakpoint triggers can be set to be executed:
  - Always
  - Once
- Breakpoints can trigger a callback when removed
- Multiple breakpoints can be set on the same address, page or opcode

## COMPLEX BREAKPOINTS

- Example:
  - Break on every MOV instruction that writes to the address range 0x00401000 – 0x00401008

```
titan_vm_set_breakpoint (process_handle,
                         &breakpoint_handle,
                         break_always_k + \
                         break_on_action_write_k + \
                         break_on_address_k + \
                         break_on_instruction_k,
                         op_mov_k,
                         0x00401000,
                         8,
                         &callback,
                         parameter);
```

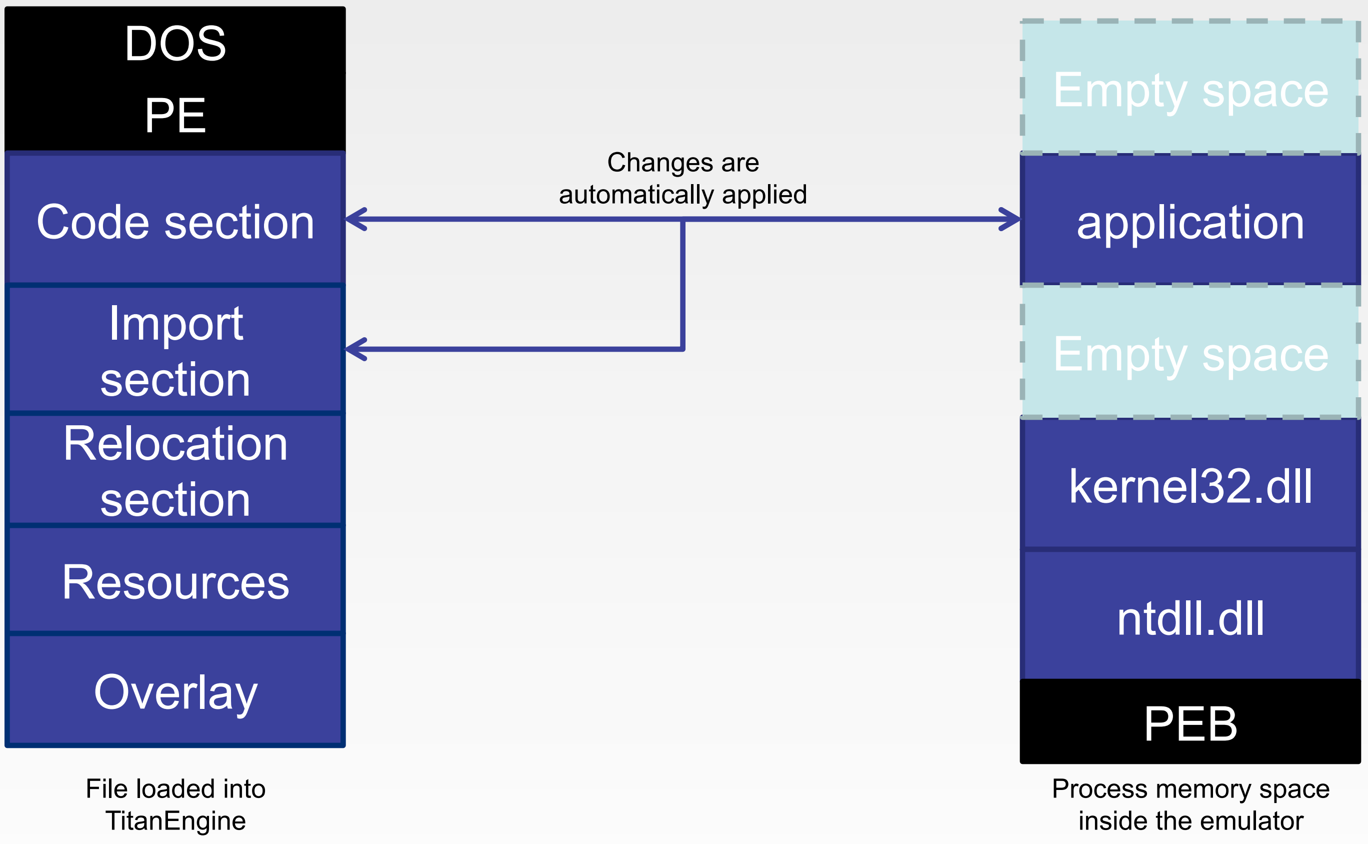
## EXCEPTIONS

- Exceptions work as you would expect them to work
- Exception chain is stored on the stack with the FS:[0] pointing to the current SEH handler
- Exceptions are passed to the emulator event callback
- Based on the return from the emulation callback the exception is either passed to the emulated process or handled
- Currently unsupported instructions will throw an exception which can be used to emulate the unsupported instruction

## ANALYSIS SYMBIOSIS

## REFLECTING CHANGES

### Files are in sync with the emulator



## EXECUTING HIGH LEVEL CODE

- Emulator callbacks provide an interface to the system API
- Every change that these APIs introduce is automatically applied to the emulated process

```
struct functions_t
{
        ...
        bool (*vm_mount_drive)(process_handle_t process_handle, ...);
        obj_handle_t (*vm_create_file)(process_handle_t process_handle, ...);
        bool (*vm_read_file)(process_handle_t process_handle, obj_handle_t file_handle, ···);
        bool (*vm_write_file)(process_handle_t process_handle, obj_handle_t file_handle, ···);
        uint32_t (*vm_set_file_position)(process_handle_t process_handle, ···);
        bool (*vm_set_end_of_file)(process_handle_t process_handle, obj_handle_t file_handle);
        uint32_t (*vm_get_file_size)(process_handle_t process_handle, obj_handle_t file_handle);
        void (*vm_close_file)(process_handle_t process_handle, obj_handle_t file_handle);
        bool (*vm_format_drive)(process_handle_t process_handle, drive_id_t drive_id);
        bool (*vm_unmount_drive)(process_handle_t process_handle, drive_id_t drive_id);
        ...
};
```

## HIGH LEVEL CODE EXAMPLE

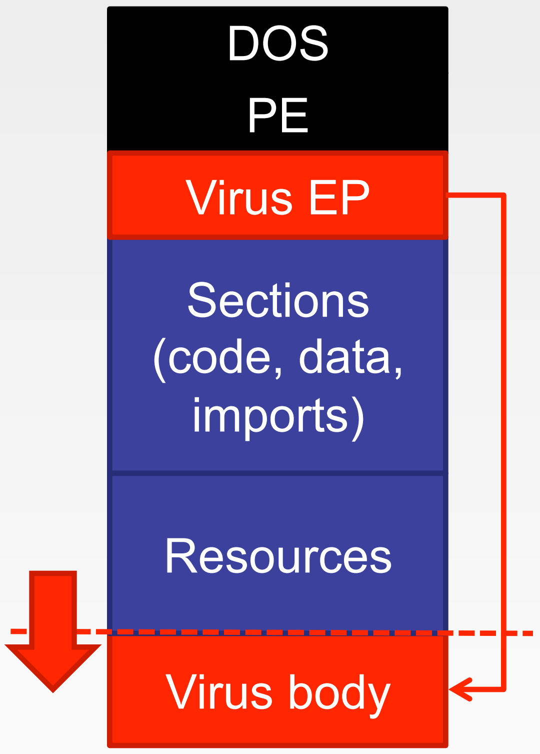
### Inserting a bait file into the file system

- Mount a new drive in the system with **vm_mount_drive**
- Install a bait file from host to emulated process with **titan_vm_import_file**
- Wait for the emulated process to infect the imported file
  - Check file characteristic changes, e.g. **vm_file_size**
- Once the file gets infected compare the differences between the infected file and the original bait
- Optionally export the infected file to host for further analysis with **titan_vm_export_file**

## DISINFECTING WIN32.VIRUT

## VIRUS.WIN32.VIRUT

### General virus information



One infection type file layout

Malware type:

- PE32 file infector
- Backdoor (IRC) / C&C
- Memory resident

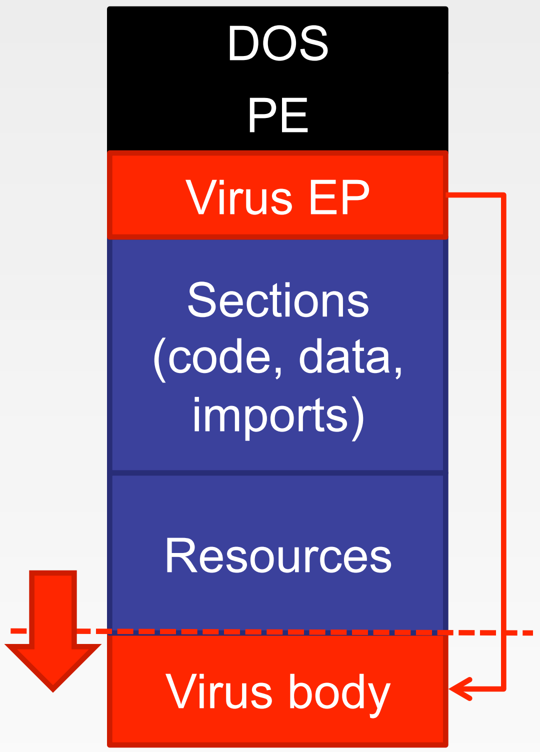
Aliases:

- Virus:W32/Virut
- Virus.Win32.Virut
- Win32.Virtob

- Versions:
  - Few different versions
  - Few different infection types

## VIRUS.WIN32.VIRUT

### Infected file behavior



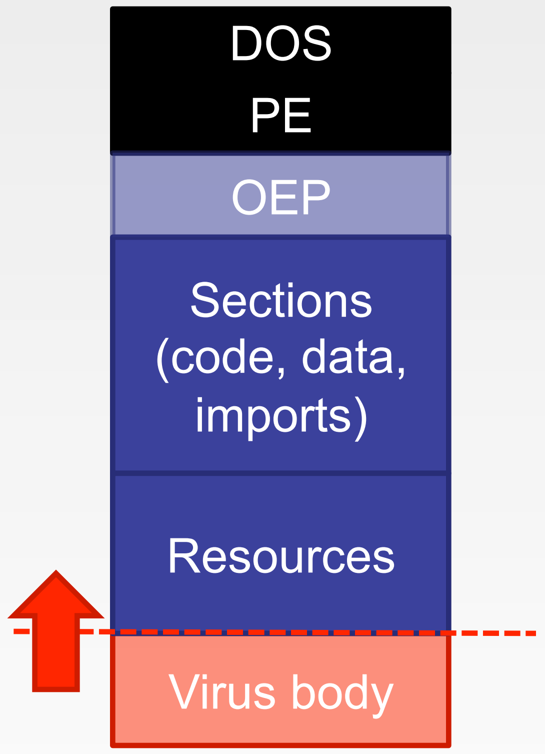
One infection type file layout

Virus body encryption

- Original entry point is moved
- One or more polymorphic layers encrypt the virus body
- Original entry point is restored from the secondary thread
- Virus performs infection in a secondary thread
- Virus embeds a UPX packed DLL that infects other PE files

## DISINFECTION PROCEDURE

### Creating disinfection routine with FDF



One infection type file layout

Disinfection procedure steps:

1. Check if the file is infected
   - Determine virus version
   - Determine if it's a virus body
2. Get the virus code delta (EBP)
3. Read the original entry point data
4. Read the original entry point address
5. Update the original entry point code
6. Resize the last section (remove virus)
7. Update the PE header with new OEP
8. Save the disinfected file

## DISINFECTION DEMO

## EMULATION STATISTICS

- Emulation runs for 5 minutes (this is a debug mode)
- During the emulation:
  - 8 dynamic link libraries are loaded
  - 245,482,988 instructions are executed
- Current average emulation speed is ~1,000 instructions/ms
- This number will tenfold once emulation is optimized
- You should **not** be running this much code but you can!

## TITANENGINE 3

**Where to get it?**

**http://titan.reversinglabs.com$^{\text{(soon)}}$**

**Future plans**

- Version 3.2
  - Porting the library to Linux
- Version 3.3
  - Implementing disassembler & assembler
  - Function analysis
- Version 3.4
  - TitanLanguage integration

REVERSING LABS | Reverse Engineering & Software Protection

## THANK YOU!

July 25, 2012