Repacking the unpacker: Applying Time Travel Debugging to malware analysis

Benoît Sevens

Exercise workbook

Exercise 1 – PeeAndGee.run

- You get a trace file of an execution of a packed malware sample in a sandbox, called `1-PeeAndGee.run`
- Can you extract the payload?

Exercise 1 – PeeAndGee.run

- Let’s start with looking for calls to `VirtualProtect(Ex)`
- Do you get any results? What does this probably mean?
- Can you find any other interesting API’s that get called?

Exercise 1 – PeeAndGee.run

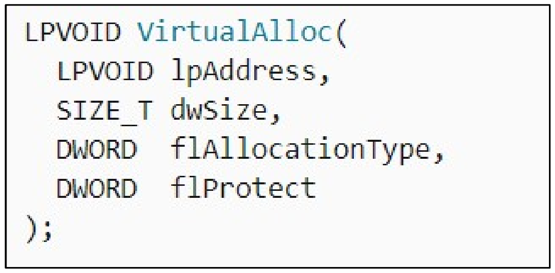
- Let's start with looking for calls to `VirtualProtect(Ex)`
  - `dx -g @$cursession.TTD.Calls("kernel32!VirtualProtect*")`
- Do you get any results? What does this probably mean?
  - Allocation of memory probably happened through one of the `VirtualAlloc(), VirtualAllocEx()` or `NtAllocateVirtualMemory()` functions.
- Can you find any other interesting API's that get called?
  - `dx -g @$cursession.TTD.Calls("kernel32!VirtualAlloc*")`

Exercise 1 – PeeAndGee.run

- What does the function prototype of `VirtualAlloc()` look like?
- Which parameter has often a specific value when unpackers allocate memory?
- Can you construct a query that only shows `VirtualAlloc()` calls with this specific parameter value?

Exercise 1 – PeeAndGee.run

- What does the function prototype of `VirtualAlloc()` look like?



```
LPVOID VirtualAlloc(
  LPVOID lpAddress,
  SIZE_T dwSize,
  DWORD  flAllocationType,
  DWORD  flProtect
);
```

- Which parameter has often a specific value when unpackers allocate memory?
  - `flProtect = 0x40`
- Can you construct a query that only shows `VirtualAlloc()` calls with this specific parameter value?
  - `dx -g @$cursession.TTD.Calls("kernel32!VirtualAlloc*").Where(c => c.Parameters[3] == 0x40)`

Exercise 1 – PeeAndGee.run

- Look at the `TimeStart` column. Do you see a problem?
- How can we correct this problem?
- Can we print a table with the size and address of each allocation?

Exercise 1 – PeeAndGee.run

- Look at the `TimeStart` column. Do you see a problem?
  - The calls are not ordered by time.
- How can we correct this problem?
  - `dx -g @$cursession.TTD.Calls("kernel32!*VirtualAlloc*").Where(c => c.Parameters[3] == 0x40).OrderBy(c => c.TimeStart)`
- Can we print a table with the size and address of each allocation?
  - `dx -g @$cursession.TTD.Calls("kernel32!*VirtualAlloc*").Where(c => c.Parameters[3] == 0x40). OrderBy(c => c.TimeStart).Select(c => new {addr = c.ReturnValue, size = c.Parameters[1]})`

Exercise 1 – PeeAndGee.run

- Is there an allocation that stands out?
- If we suppose this section contains the unpacked code, what will happen after the unpacked bytes get written there?
- How can we find the first time an instruction gets executed in this part of memory?

Exercise 1 – PeeAndGee.run

- Is there an allocation that stands out?
  - There is one that has a different size.
- If we suppose this section contains the unpacked code, what will happen after the unpacked bytes get written there?
  - The code will eventually jump/call into it, so it will get executed.
- How can we find the first time an instruction gets executed in this part of memory?
  - `dx @$cursession.TTD.Memory(0x1c070000 , 0x1c07c000, "e").OrderBy(m => m.TimeStart)[0]`

Exercise 1 – PeeAndGee.run

- If we time travel to this instruction and step one instruction back, we should see the instruction that jumped/called into the unpacked code. Which one is that?
- At this moment in time, we can hope for a MZ header at the beginning of the allocated memory. Is this the case?
- What WinDbg extension command can print some header information about this executable?

Exercise 1 – PeeAndGee.run

- If we time travel to this instruction and step one instruction back, we should see the instruction that jumped/called into the unpacked code. Which one is that?
  - `!tt 35F9F:43`
  - `t-`
- At this moment in time, we can hope for a MZ header at the beginning of the allocated memory. Is this the case?
  - `db 0x1c070000`
- What WinDbg extension command can print some header information about this executable?
  - `!dh 0x1c070000`

Exercise 1 – PeeAndGee.run

- Is the PE file in memory “memory mapped”?
- What is the size of the executable on disk? You can deduce this from information of the last section.
- Can you dump the relevant memory area to disk?

Exercise 1 – PeeAndGee.run

- Is the PE file in memory “memory mapped”?
  - The first section starts at offset 0x400 on disk and at offset 0x1000 in memory. Because there is data at 0x1c070000 + 0x400, the PE file is as it would be on disk.
- What is the size of the executable on disk? You can deduce this from information of the last section.
  - The last section starts at offset 0x3400 and has a size of 0x400. The size of the executable is thus 0x3800 bytes.
- Can you dump the relevant memory area to disk?
  - `.writemem 1-PeeAndGee.bin 1c070000 L3800`

Exercise 1 – PeeAndGee.run

- **Bonus question:** Why did the malware allocate 0xc000 bytes, while it only takes 0x3800 bytes on disk?

Exercise 1 – PeeAndGee.run

- **Bonus question:** Why did the malware allocate 0xc000 bytes, while it only takes 0x3800 bytes on disk?
  - The file as it is on disk is not usable in memory. It needs to be mapped to memory. Where did it get mapped? From the header info, we see that the entry point is at offset 0x1b40 from the image base. The unpacker jumped to address 0x1c077b40, which means that the memory mapped executable starts at address 0x1c076000. If we check the memory content there (db 0x1c076000), we indeed find a MZ header. So in summary:
    - 0x1c070000 - 0x1c073800: executable laid out as it is on disk
    - 0x1c076000 - 0x1c07c000: executable mapped in memory

Exercise 2 – Reflect.run

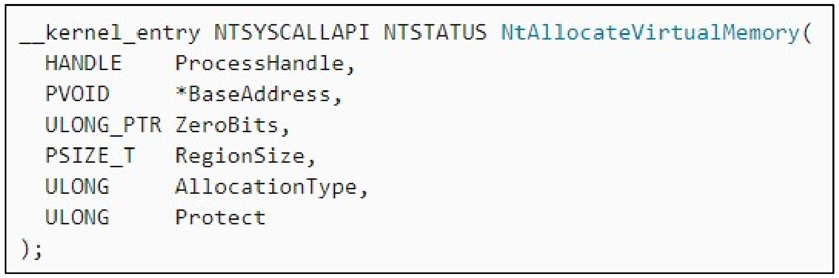
- Same question, different sample…
- `2-Reflect.run`

Exercise 2 – Reflect.run

- What interesting memory allocation calls do you find in this sample?
- What is the prototype of this function?
- Inspect the parameters of each call. Do you see anything interesting/suspicious?

Exercise 2 – Reflect.run

- What interesting memory allocation calls do you find in this sample?
  - `dx -g @$cursession.TTD.Calls("ntdll!NtAllocateVirtualMemory")`
- What is the prototype of this function?



```
__kernel_entry NTSYSCALLAPI NTSTATUS NtAllocateVirtualMemory(
  HANDLE    ProcessHandle,
  PVOID     *BaseAddress,
  ULONG_PTR ZeroBits,
  PSIZE_T   RegionSize,
  ULONG     AllocationType,
  ULONG     Protect
);
```

- Inspect the parameters of each call. Do you see an interesting/suspicious call?
  - `dx -g @$cursession.TTD.Calls("ntdll!NtAllocateVirtualMemory").OrderBy(c => c.TimeStart).Select(c => c.Parameters)`

Exercise 2 – Reflect.run

- What is the requested memory protections of the allocated memory area?
- What API call would typically follow such behavior if the malware is unpacking?
- Can you find calls to this API in the trace?

Exercise 2 – Reflect.run

- What is the requested memory protections of the allocated memory area?
  - `0x40 (PAGE_EXECUTE_READWRITE)`
- What API call would typically follow such behavior if the malware is unpacking?
  - `WriteProcessMemory()` or `NtWriteVirtualMemory()`
- Can you find calls to this API in the trace? What are the parameters provided?

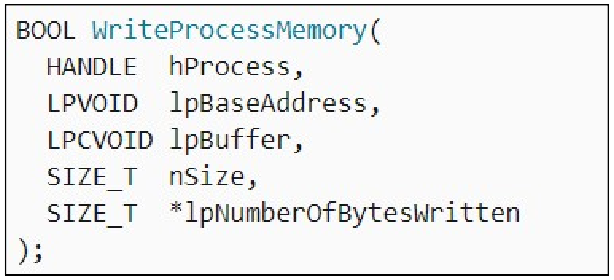
  -
    ```
    dx -g
    @$cursession.TTD.Calls("kernel32!WriteProcessMemory*").OrderBy(c => c.TimeStart).Select(c => c.Parameters)
    ```

Exercise 2 – Reflect.run

- What is the function prototype for this API call?
- Inspect the function calls and their parameters, knowing their meaning. Do you have an assumption of what happened?
- Go to the first function call, inspect the buffer contents and confirm your assumption.

Exercise 2 – Reflect.run

- What is the function prototype for this API call?



```
BOOL WriteProcessMemory(
  HANDLE  hProcess,
  LPVOID  lpBaseAddress,
  LPCVOID lpBuffer,
  SIZE_T  nSize,
  SIZE_T  *lpNumberOfBytesWritten
);
```

- Inspect the function calls and their parameters. Do you have an assumption of what happened?
  - The sample injected code into another process piece by piece.
- Go to the first function call, inspect the buffer contents and confirm your assumption.
  - `dx -g @$cursession.TTD.Calls("kernel32!WriteProcessMemory*").OrderBy(c => c.TimeStart)`
  - `!tt 637:5C2`
  - `db @r8`

Exercise 2 – Reflect.run

- Describe in your own words how you would dump this sample.
- Doing this by hand would be quite tedious and error prone. This is a good example of a scenario where JavaScript automation can be helpful. A skeleton of a script that does this is provided in `2-Reflect.js`. Can you fill in the missing pieces and dump the sample?

Exercise 2 – Reflect.run

- Describe in your own words how you would dump this sample.
  - Time travel to each call to `WriteProcessMemory()` and dump the buffer that is about to be injected into a file.
- Doing this by hand would be quite tedious and error prone. This is a good example of a scenario where JavaScript automation can be helpful. A skeleton of a script that does this is provided in `2-Reflect.js`. Can you fill in the missing pieces and dump the sample?
  - No help here ☺

Exercise 2 – Reflect.run

- **Bonus question:** What does the payload you dumped do exactly? A little bit of reverse engineering can give you the answer. A flag is to be found!