SUSE

20 APRIL 2021

Reverse engineering the Windows SMB server

Aurélien Aptel <aaptel@suse.com>

Reverse engineering the Windows SMB server

"Reverse engineering is taking apart an object to see how it works in order to duplicate or enhance the object."

- Why?
  - Dump cryptographic keys generated by the SMB server used for encryption
  - Fun?
- Useful for:
  - Debugging while implementing SMB encryption
  - Decrypting a network capture in Wireshark

Plan

- Windows kernel, differences and comparaison with Linux kernel
- Finding the code for the SMB server
- WinDbg and Windows kernel debugging
- Disassemblers and static analysis tools, IDA pro
- When and where the SMB server generates keys
- Ways to automaticaly dump the key as it gets generated
- Summary of the implemented solution
- Final words

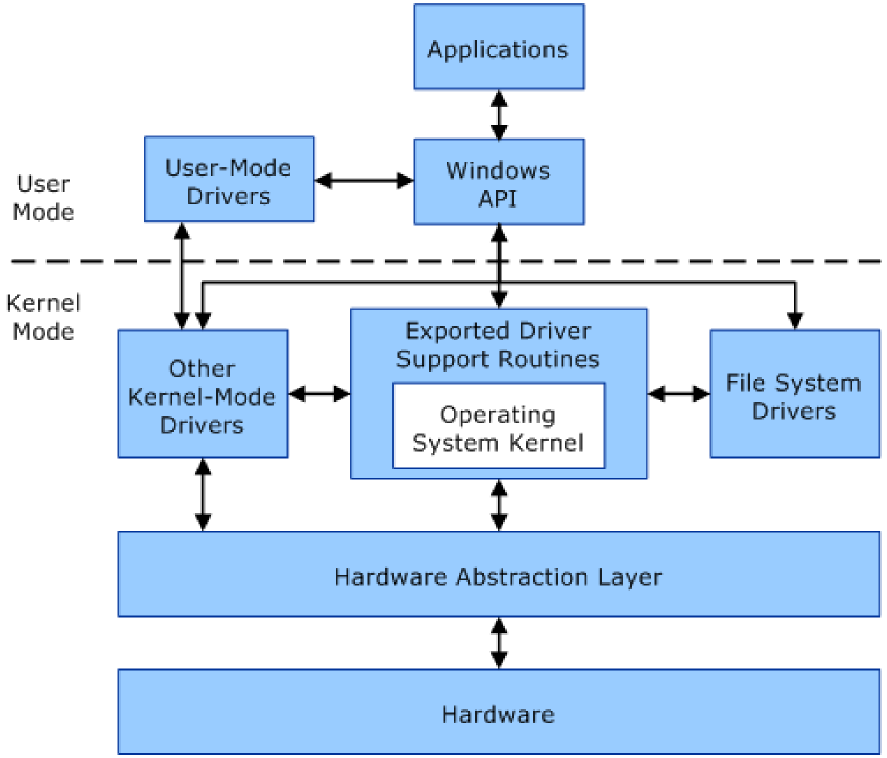
Dumping SMB traffic pre-encryption

If you are only interested in the decrypted traffic and not the keys: this is already possible (thx Obaid!)

- On client
  - `netsh trace start provider=Microsoft-Windows-SMBClient capture=yes`
- On server
  - `netsh trace start provider= Microsoft-Windows-SMBServer capture=yes`
- To stop trace and generate the .etl file
  - `netsh trace stop`
- To convert ETL to pcap https://github.com/microsoft/etl2pcapng
- https://channel9.msdn.com/events/Open-Specifications-Plugfests/Redmond-Interoperability-Protocols-Plugfest-2015/Decrypting-SMB3-Protocol

Overview of the Windows SMB server

SMB server is implemented as kernel modules (drivers in Windows jargon)



Overview of the Windows SMB server

- Most drivers are stored in `%SystemRoot%\system32\drivers\`
- Drivers use the `.sys` extension
- Use the PE file header
  - Same as .exe or .dll

Overview of the Windows SMB server

Kernel modules	Windows	Linux
Location	`C:\Windows\System32\drivers`	`/lib/modules/$version/`
Extension	`.sys`	`.ko`
File format	PE	ELF

Overview of the Windows SMB server

- Where is the server?

- First attempt: look for "smb2" occurrences in all the drivers

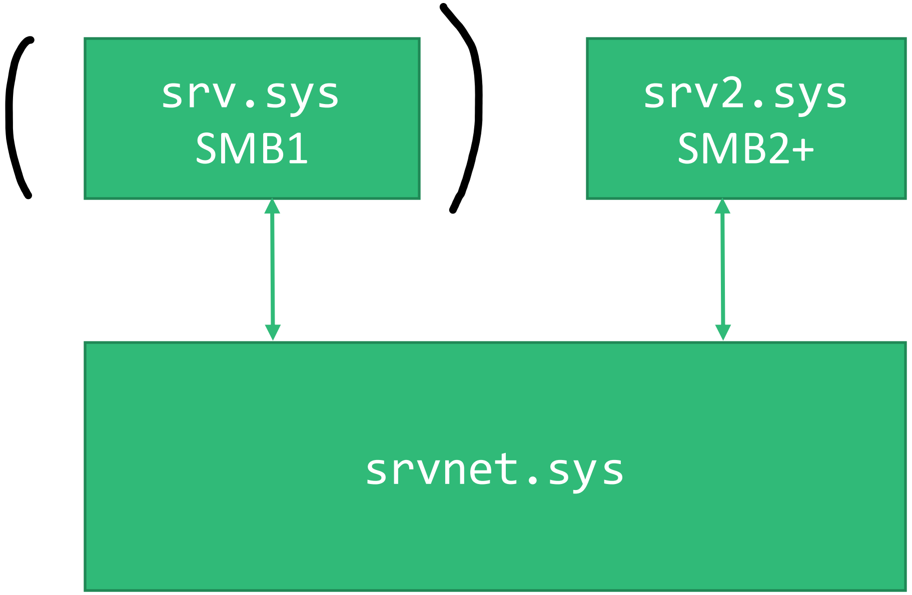
```
$ strings --print-file-name -n 8 *.sys | grep -i smb2
mrxsmb20.sys: ...
mrxsmb.sys: ...
srv2.sys: ...
srvnet.sys: ...
```

- mrxsmb* : SMB redirectors (client)

- srv* : SMB server!

Overview of the Windows SMB server

The SMB server implementation seems to be done in mainly 3 drivers

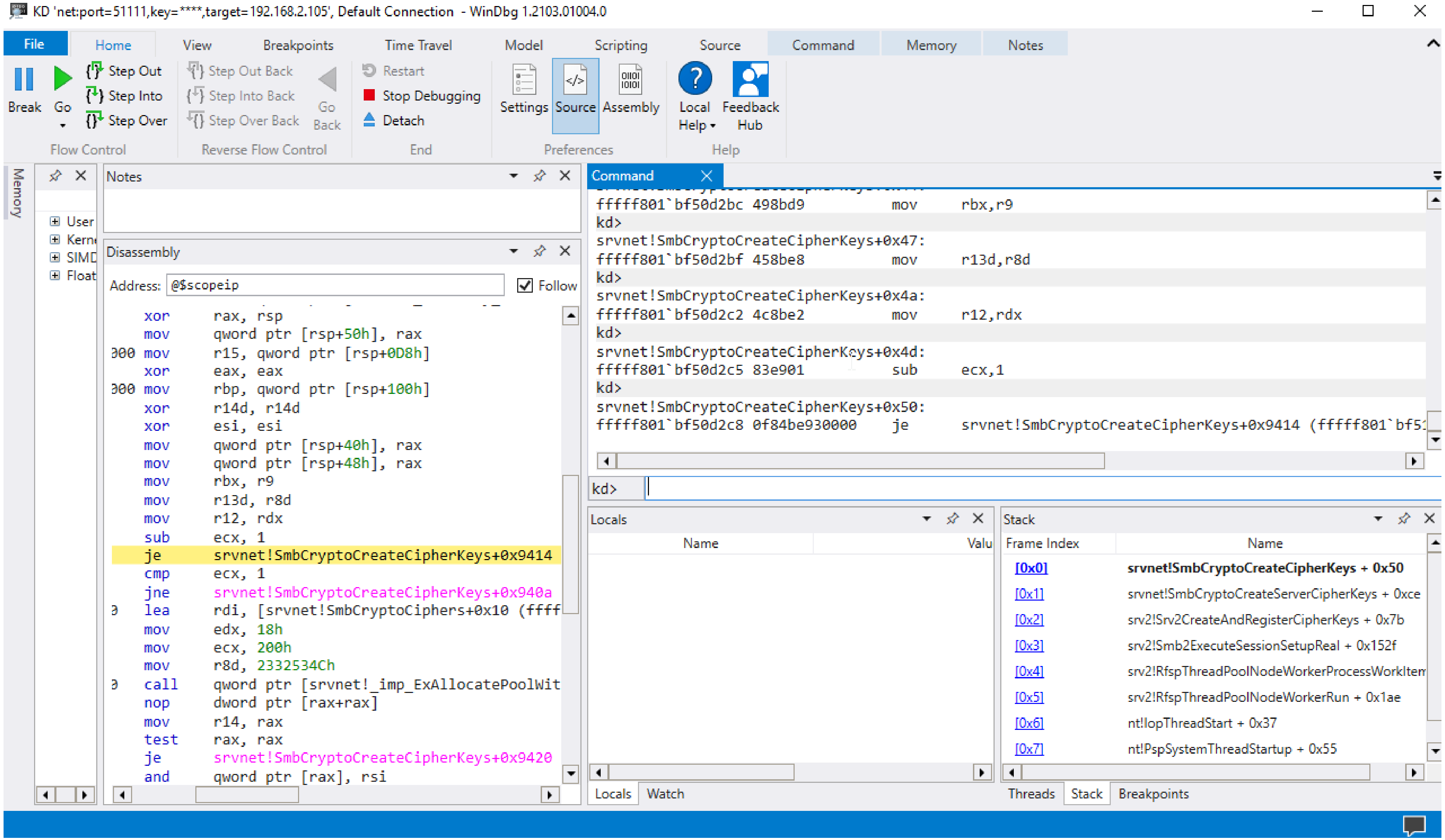


Debugger

Microsoft has an official stand-alone debugger called WinDbg

- Userspace debugging
- Remote debugging (kernel or userspace)
- Rudimentary GUI with a command-line interface
  - Pure Text also possible (cdb, kd)
- Incompatible with GDB
- WinDbg "Preview"
  - More modern GUI wrapper

Debugger



KD 'net:port=51111,key=****,target=192.168.2.105', Default Connection - WinDbg 1.2103.01004.0

Address: @$scopeip ☑ Follow

```
    xor     rax, rsp
    mov     qword ptr [rsp+50h], rax
300 mov     r15, qword ptr [rsp+0D8h]
    xor     eax, eax
300 mov     rbp, qword ptr [rsp+100h]
    xor     r14d, r14d
    xor     esi, esi
    mov     qword ptr [rsp+40h], rax
    mov     qword ptr [rsp+48h], rax
    mov     rbx, r9
    mov     r13d, r8d
    mov     r12, rdx
    sub     ecx, 1
    je      srvnet!SmbCryptoCreateCipherKeys+0x9414
    cmp     ecx, 1
    jne     srvnet!SmbCryptoCreateCipherKeys+0x940a
3   lea     rdi, [srvnet!SmbCryptoCiphers+0x10 (ffff
    mov     edx, 18h
    mov     ecx, 200h
    mov     r8d, 2332534Ch
3   call    qword ptr [srvnet!_imp_ExAllocatePoolWit
    nop     dword ptr [rax+rax]
    mov     r14, rax
    test    rax, rax
    je      srvnet!SmbCryptoCreateCipherKeys+0x9420
    and     qword ptr [rax], rsi
```

```
fffff801`bf50d2bc 498bd9          mov     rbx,r9
kd>
srvnet!SmbCryptoCreateCipherKeys+0x47:
fffff801`bf50d2bf 458be8          mov     r13d,r8d
kd>
srvnet!SmbCryptoCreateCipherKeys+0x4a:
fffff801`bf50d2c2 4c8be2          mov     r12,rdx
kd>
srvnet!SmbCryptoCreateCipherKeys+0x4d:
fffff801`bf50d2c5 83e901          sub     ecx,1
kd>
srvnet!SmbCryptoCreateCipherKeys+0x50:
fffff801`bf50d2c8 0f84be930000    je      srvnet!SmbCryptoCreateCipherKeys+0x9414 (fffff801`bf5
```

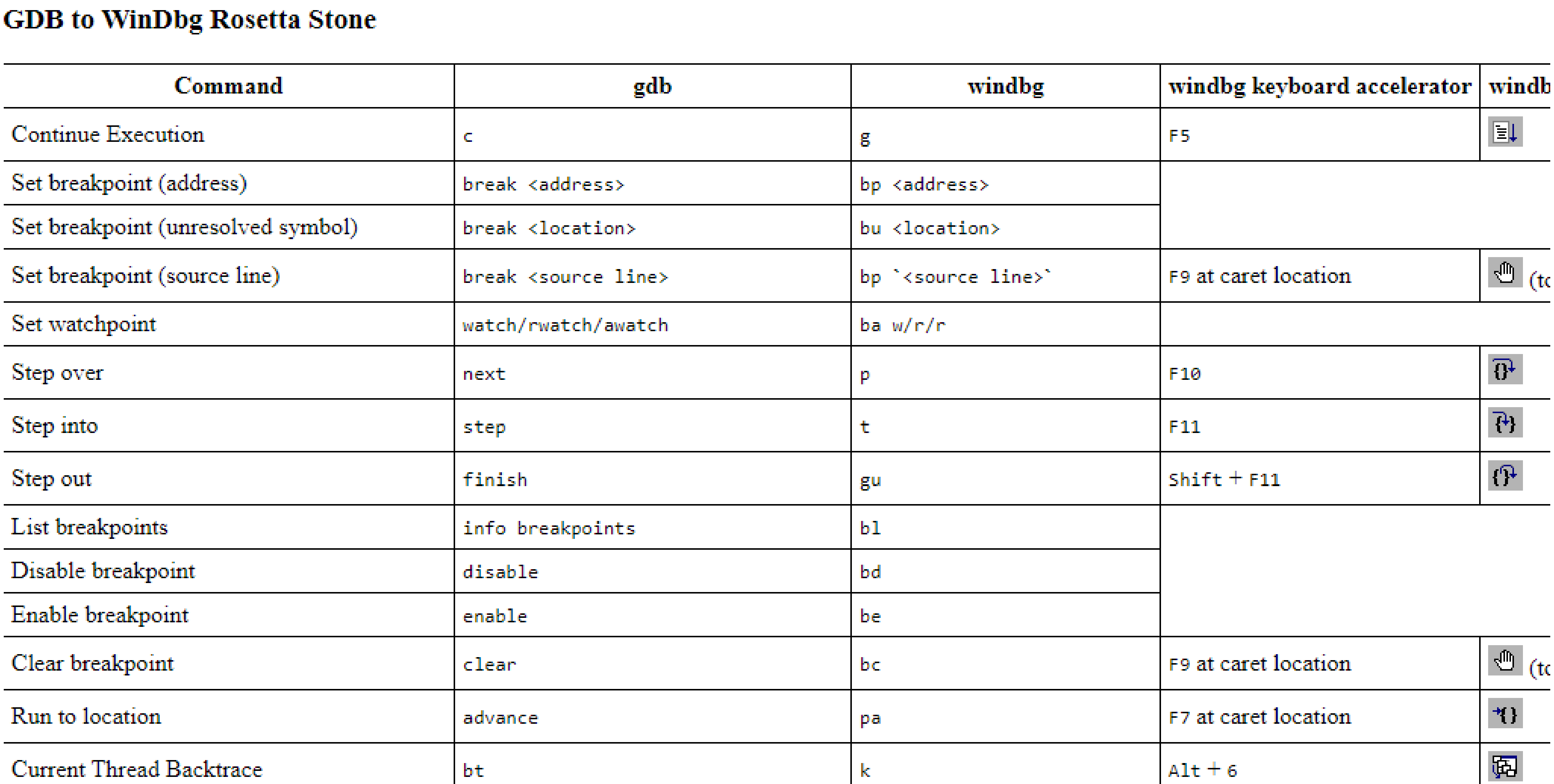
kd>

Stack

Frame Index	Name
[0x0]	**srvnet!SmbCryptoCreateCipherKeys + 0x50**
[0x1]	srvnet!SmbCryptoCreateServerCipherKeys + 0xce
[0x2]	srv2!Srv2CreateAndRegisterCipherKeys + 0x7b
[0x3]	srv2!Smb2ExecuteSessionSetupReal + 0x152f
[0x4]	srv2!RfspThreadPoolNodeWorkerProcessWorkItem
[0x5]	srv2!RfspThreadPoolNodeWorkerRun + 0x1ae
[0x6]	nt!IopThreadStart + 0x37
[0x7]	nt!PspSystemThreadStartup + 0x55

Debugger

"*Aaron's shitty windbg cheat sheet*" from https://dblohm7.ca/pmo/windbgcheatsheet.html



GDB to WinDbg Rosetta Stone

Command	gdb	windbg	windbg keyboard accelerator	windb
Continue Execution	`c`	`g`	`F5`	
Set breakpoint (address)	`break <address>`	`bp <address>`		
Set breakpoint (unresolved symbol)	`break <location>`	`bu <location>`		
Set breakpoint (source line)	`break <source line>`	``bp `<source line>` ``	`F9` at caret location	(tc
Set watchpoint	`watch/rwatch/awatch`	`ba w/r/r`		
Step over	`next`	`p`	`F10`	
Step into	`step`	`t`	`F11`	
Step out	`finish`	`gu`	`Shift` + `F11`	
List breakpoints	`info breakpoints`	`bl`		
Disable breakpoint	`disable`	`bd`		
Enable breakpoint	`enable`	`be`		
Clear breakpoint	`clear`	`bc`	`F9` at caret location	(tc
Run to location	`advance`	`pa`	`F7` at caret location	
Current Thread Backtrace	`bt`	`k`	`Alt` + `6`	

Debugger

How to debug the kernel?

- Dual machine setup
  - Host is running WinDbg, waiting for connections
  - Debugged target (VM for me) is configured for remote debugging, connects to host
  - https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/setting-up-a-network-debugging-connection-automatically
- Note: target requires a supported NIC! Pick virtual NIC model carefully...
  - Silently fails with qemu virtio NIC... Even qemu intel E100... :(

Debugger

How to debug the kernel?

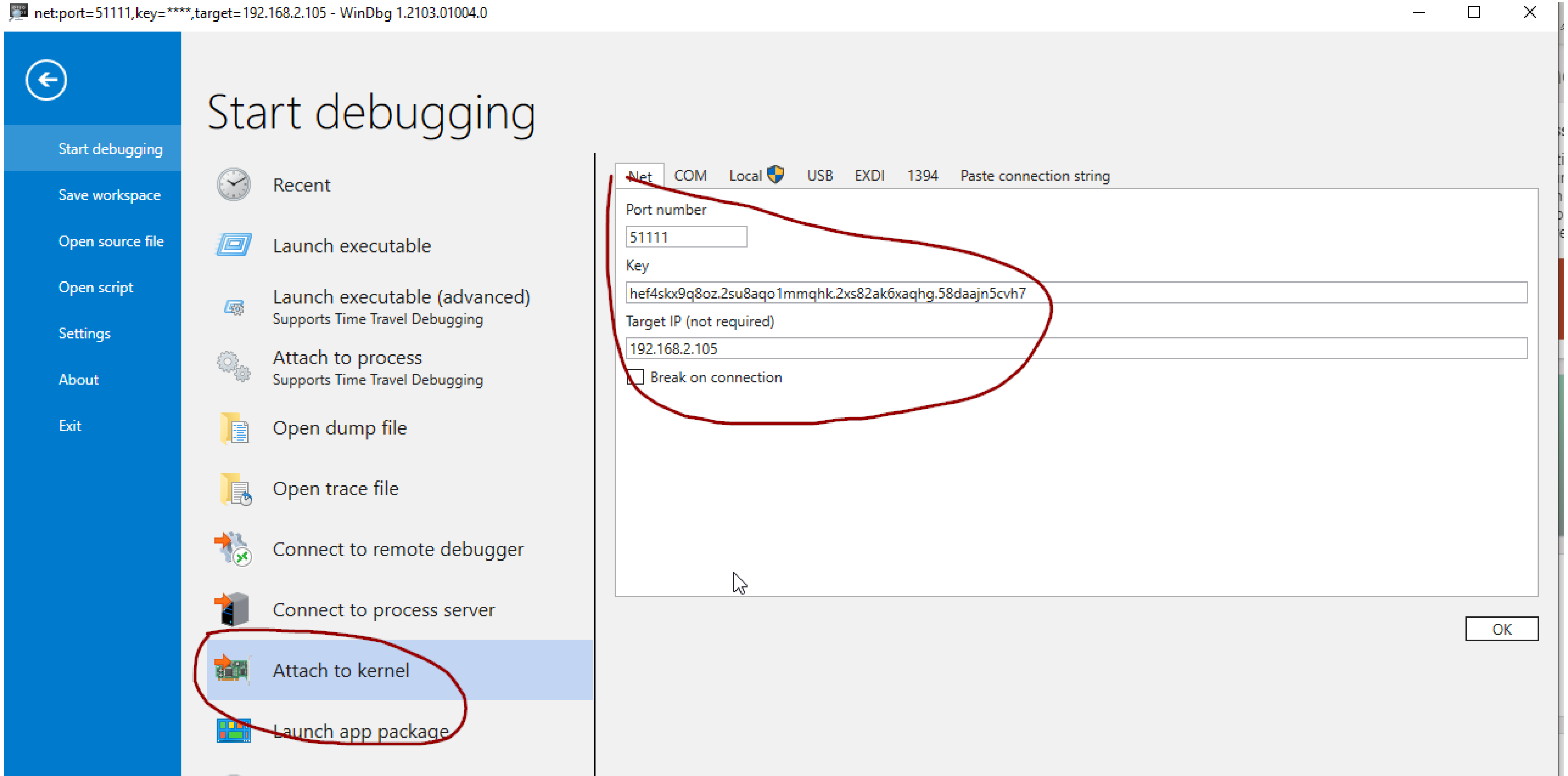
- Setting up the target: install the debugging tools then

```
>kdnet.exe <HostComputerIPAddress> <YourDebugPort>
Enabling network debugging on Intel(R) 82577LM Gigabit Network Connection.
Key=2steg4fzbj2sz.23418vzkd4ko3.1g34ou07z4pev.1sp3yo9yz874p
```

- In remote network debugging, host creates a debug TCP server, target connects to it
  - Similar to FTP active mode (kind of backward)
- Special debug boot mode enabled by default
  - Can list boot config with `bcdedit /dbgsettings`
- Reboot

Debugger

- Setting up the host
  - Just start WinDbg with the Key and Port from the target (command line or GUI)



net:port=51111,key=****,target=192.168.2.105 - WinDbg 1.2103.01004.0

Start debugging

Port number: 51111

Key: hef4skx9q8oz.2su8aqo1mmqhk.2xs82ak6xaqhg.58daajn5cvh7

Target IP (not required): 192.168.2.105

□ Break on connection

Debugger

- Survival guide edition

`.reload`	Refresh loaded symbols
`lm`	List loaded kernel modules
`x srv2!*key*`	List symbols containing 'key' in the srv2 module

`db expr`
`dd expr`
`dq expr`
	Hexdump of "expr", displayed as bytes (b), double word (d, 32bits), quad (q, 64bits)

`p` `t`	Step over Step into (for call instructions)

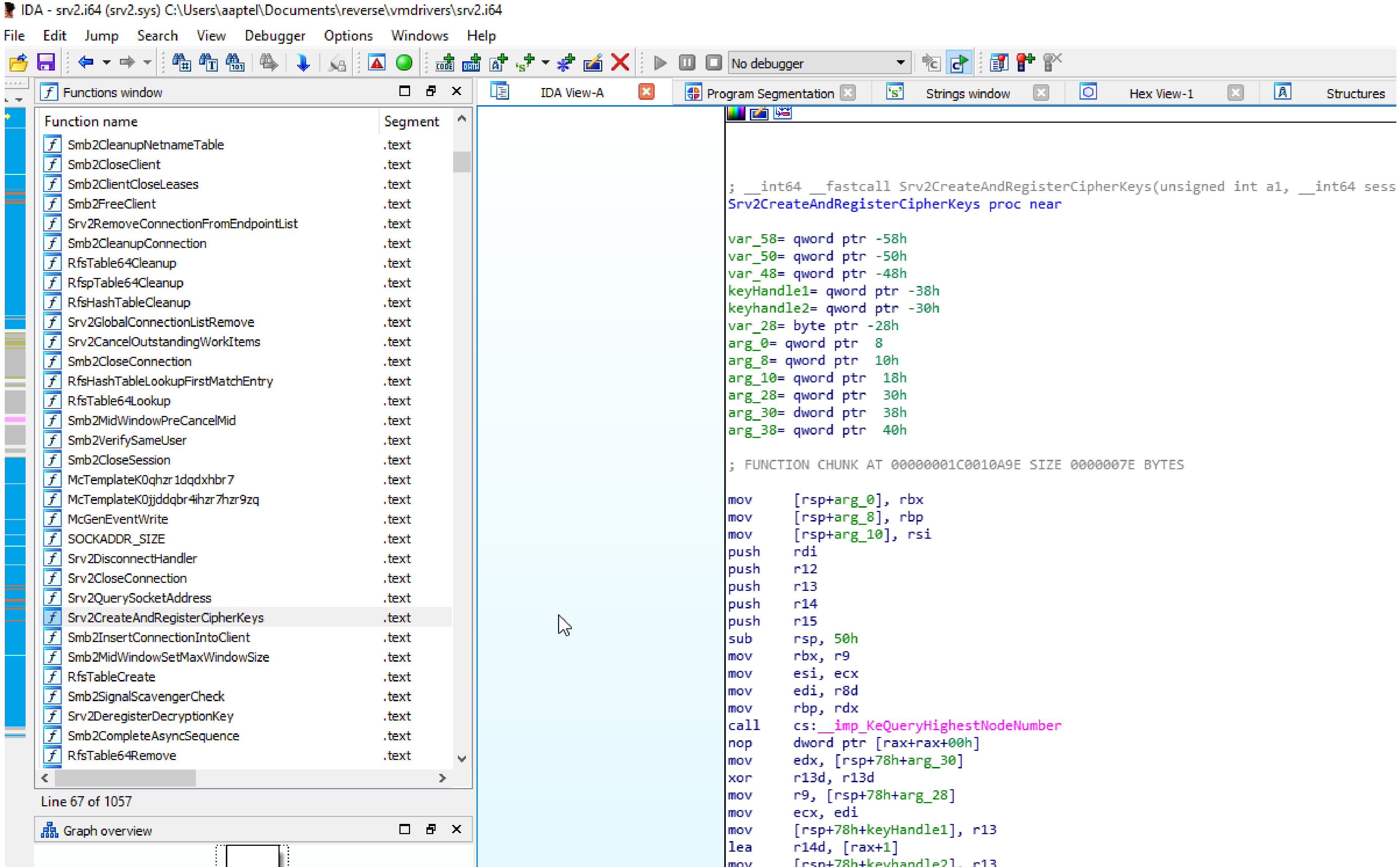
`bp expr`
`g`
	Set breakpoint on expr (addr, symbol, symbol+addr, …)
	Continue

Disassembler/Static analysis

- Tools to look around binary files
  - Import tables, export tables, disassembly, decompilation, xref, list strings, etc

- Most popular ones
  - **IDA Pro**: industry leader, closed source, expensive (but free & demo versions available)
  - Ghidra: recent, developed by the NSA, open source
  - Radare2: open source, Linux only, command-line
  - x64dbg: open source, Windows only
  - OllyDbg: freeware, Windows only, popular but old

srv2.sys in IDA

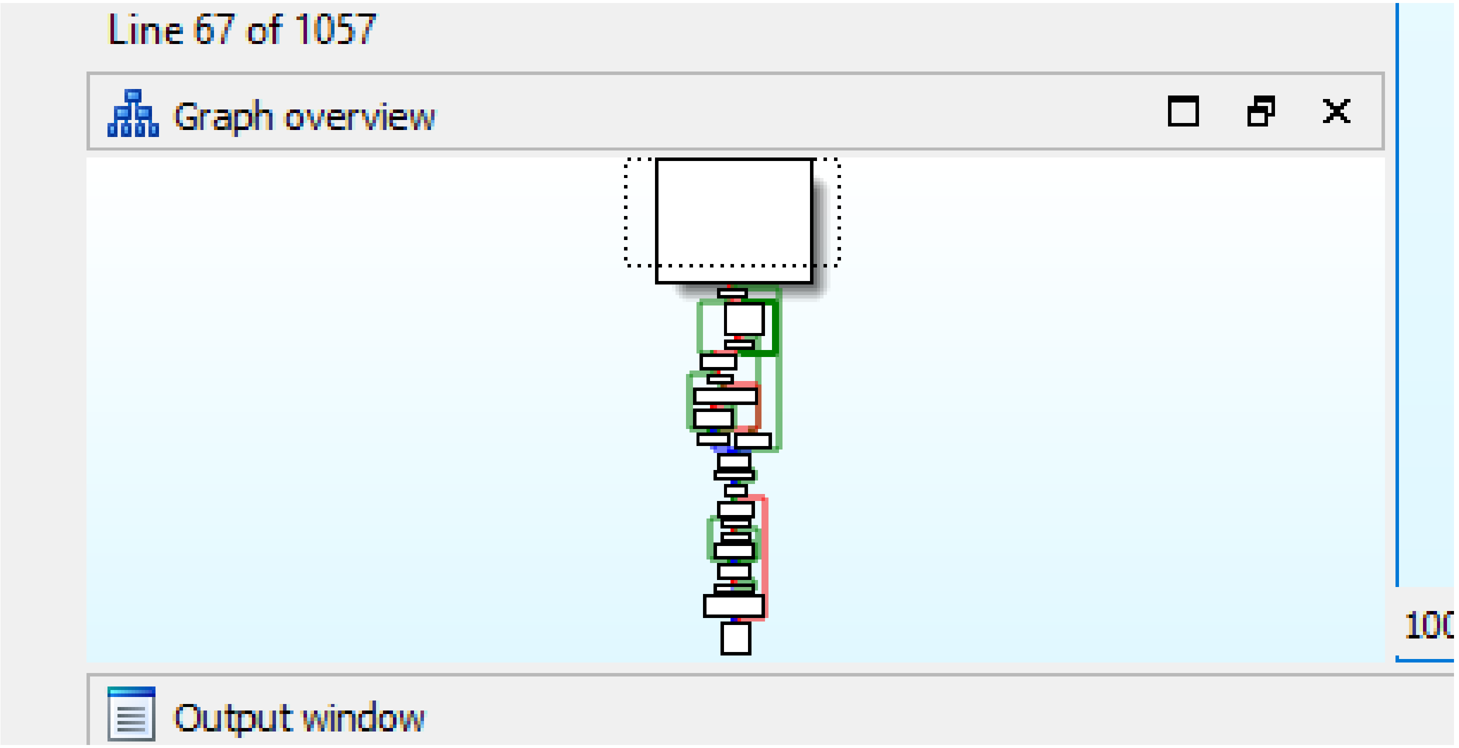


Function name	Segment
Smb2CleanupNetnameTable	.text
Smb2CloseClient	.text
Smb2ClientCloseLeases	.text
Smb2FreeClient	.text
Srv2RemoveConnectionFromEndpointList	.text
Smb2CleanupConnection	.text
RfsTable64Cleanup	.text
RfspTable64Cleanup	.text
RfsHashTableCleanup	.text
Srv2GlobalConnectionListRemove	.text
Srv2CancelOutstandingWorkItems	.text
Smb2CloseConnection	.text
RfsHashTableLookupFirstMatchEntry	.text
RfsTable64Lookup	.text
Smb2MidWindowPreCancelMid	.text
Smb2VerifySameUser	.text
Smb2CloseSession	.text
McTemplateK0qhzr1dqdxhbr7	.text
McTemplateK0jjddqbr4ihzr7hzr9zq	.text
McGenEventWrite	.text
SOCKADDR_SIZE	.text
Srv2DisconnectHandler	.text
Srv2CloseConnection	.text
Srv2QuerySocketAddress	.text
Srv2CreateAndRegisterCipherKeys	.text
Smb2InsertConnectionIntoClient	.text
Smb2MidWindowSetMaxWindowSize	.text
RfsTableCreate	.text
Smb2SignalScavengerCheck	.text
Srv2DeregisterDecryptionKey	.text
Smb2CompleteAsyncSequence	.text
RfsTable64Remove	.text

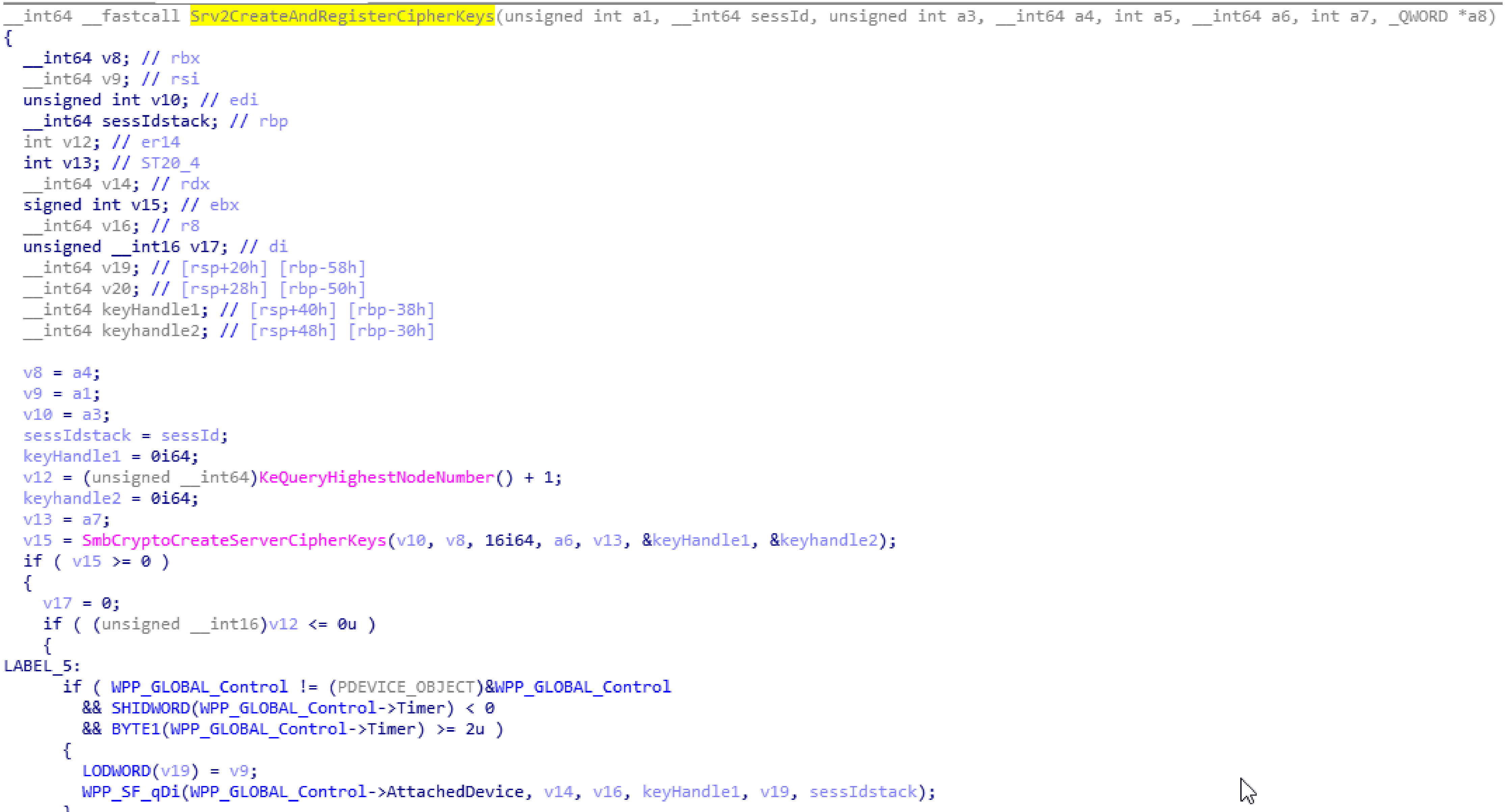
```
; __int64 __fastcall Srv2CreateAndRegisterCipherKeys(unsigned int a1, __int64 sess
Srv2CreateAndRegisterCipherKeys proc near

var_58= qword ptr -58h
var_50= qword ptr -50h
var_48= qword ptr -48h
keyHandle1= qword ptr -38h
keyhandle2= qword ptr -30h
var_28= byte ptr -28h
arg_0= qword ptr  8
arg_8= qword ptr  10h
arg_10= qword ptr  18h
arg_28= qword ptr  30h
arg_30= dword ptr  38h
arg_38= qword ptr  40h

; FUNCTION CHUNK AT 00000001C0010A9E SIZE 0000007E BYTES

mov     [rsp+arg_0], rbx
mov     [rsp+arg_8], rbp
mov     [rsp+arg_10], rsi
push    rdi
push    r12
push    r13
push    r14
push    r15
sub     rsp, 50h
mov     rbx, r9
mov     esi, ecx
mov     edi, r8d
mov     rbp, rdx
call    cs:__imp_KeQueryHighestNodeNumber
nop     dword ptr [rax+rax+00h]
mov     edx, [rsp+78h+arg_30]
xor     r13d, r13d
mov     r9, [rsp+78h+arg_28]
mov     ecx, edi
mov     [rsp+78h+keyHandle1], r13
lea     r14d, [rax+1]
mov     [rsp+78h+keyhandle2], r13
```

Srv2.sys in IDA

- Graph view



Decompiling a function in IDA

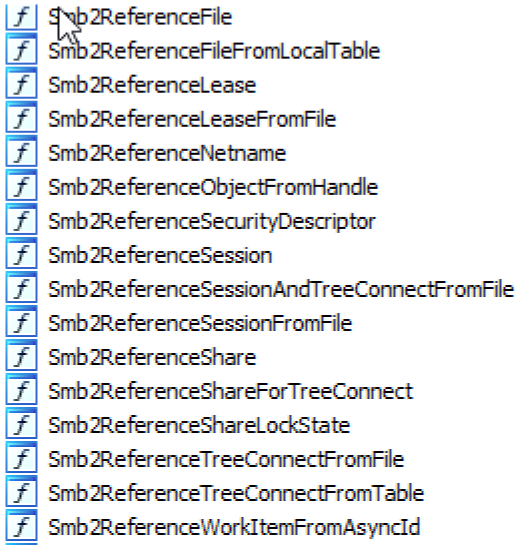


```
__int64 __fastcall Srv2CreateAndRegisterCipherKeys(unsigned int a1, __int64 sessId, unsigned int a3, __int64 a4, int a5, __int64 a6, int a7, _QWORD *a8)
{
  __int64 v8; // rbx
  __int64 v9; // rsi
  unsigned int v10; // edi
  __int64 sessIdstack; // rbp
  int v12; // er14
  int v13; // ST20_4
  __int64 v14; // rdx
  signed int v15; // ebx
  __int64 v16; // r8
  unsigned __int16 v17; // di
  __int64 v19; // [rsp+20h] [rbp-58h]
  __int64 v20; // [rsp+28h] [rbp-50h]
  __int64 keyHandle1; // [rsp+40h] [rbp-38h]
  __int64 keyhandle2; // [rsp+48h] [rbp-30h]

  v8 = a4;
  v9 = a1;
  v10 = a3;
  sessIdstack = sessId;
  keyHandle1 = 0i64;
  v12 = (unsigned __int64)KeQueryHighestNodeNumber() + 1;
  keyhandle2 = 0i64;
  v13 = a7;
  v15 = SmbCryptoCreateServerCipherKeys(v10, v8, 16i64, a6, v13, &keyHandle1, &keyhandle2);
  if ( v15 >= 0 )
  {
    v17 = 0;
    if ( (unsigned __int16)v12 <= 0u )
    {
LABEL_5:
      if ( WPP_GLOBAL_Control != (PDEVICE_OBJECT)&WPP_GLOBAL_Control
        && SHIDWORD(WPP_GLOBAL_Control->Timer) < 0
        && BYTE1(WPP_GLOBAL_Control->Timer) >= 2u )
      {
        LODWORD(v19) = v9;
        WPP_SF_qDi(WPP_GLOBAL_Control->AttachedDevice, v14, v16, keyHandle1, v19, sessIdstack);
      }
```

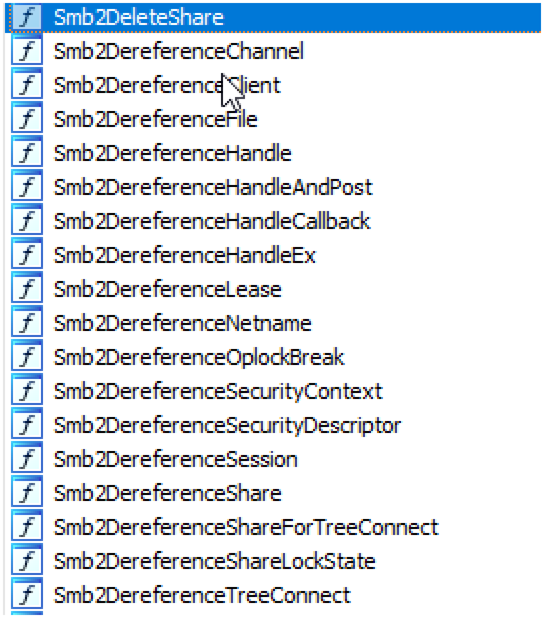
Deducting types and objects

- Certain objects can be deducted from looking at function names
  - Reference and Dereference funcs are used to keep track of reference counts (inc/dec)
  - They all must take the object pointer as parameter
  - We can figure out the offset and size of the refcount field from them

- **By iteratively annotating the prototype of the functions, the IDA decompiler can deduct and propagate more types, intermediary variable, and new func prototypes**



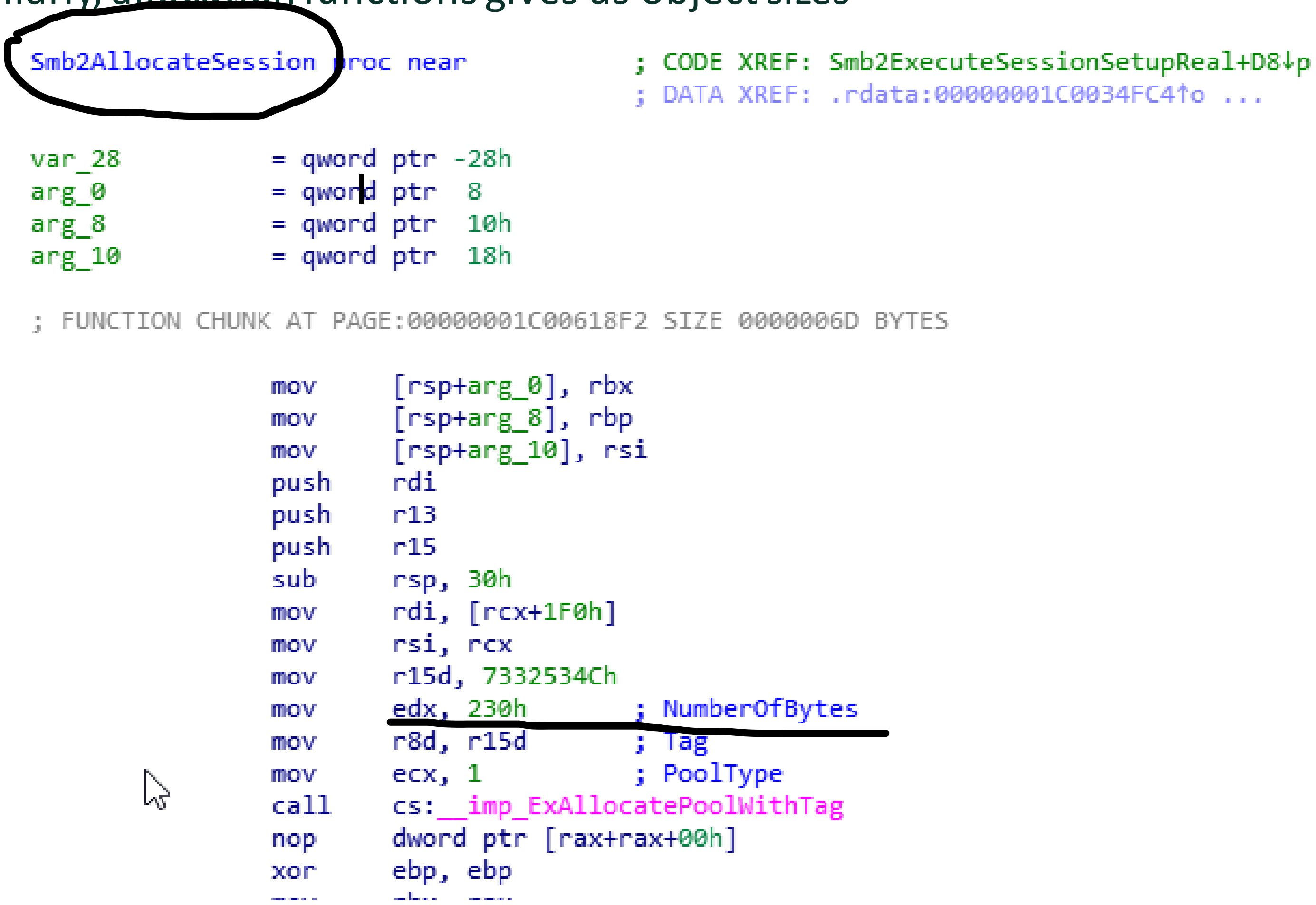
Smb2ReferenceFile
Smb2ReferenceFileFromLocalTable
Smb2ReferenceLease
Smb2ReferenceLeaseFromFile
Smb2ReferenceNetname
Smb2ReferenceObjectFromHandle
Smb2ReferenceSecurityDescriptor
Smb2ReferenceSession
Smb2ReferenceSessionAndTreeConnectFromFile
Smb2ReferenceSessionFromFile
Smb2ReferenceShare
Smb2ReferenceShareForTreeConnect
Smb2ReferenceShareLockState
Smb2ReferenceTreeConnectFromFile
Smb2ReferenceTreeConnectFromTable
Smb2ReferenceWorkItemFromAsyncId



Smb2DeleteShare
Smb2DereferenceChannel
Smb2DereferenceClient
Smb2DereferenceFile
Smb2DereferenceHandle
Smb2DereferenceHandleAndPost
Smb2DereferenceHandleCallback
Smb2DereferenceHandleEx
Smb2DereferenceLease
Smb2DereferenceNetname
Smb2DereferenceOplockBreak
Smb2DereferenceSecurityContext
Smb2DereferenceSecurityDescriptor
Smb2DereferenceSession
Smb2DereferenceShare
Smb2DereferenceShareForTreeConnect
Smb2DereferenceShareLockState
Smb2DereferenceTreeConnect

Deducting types and objects

- Similarly, allocation functions gives us object sizes

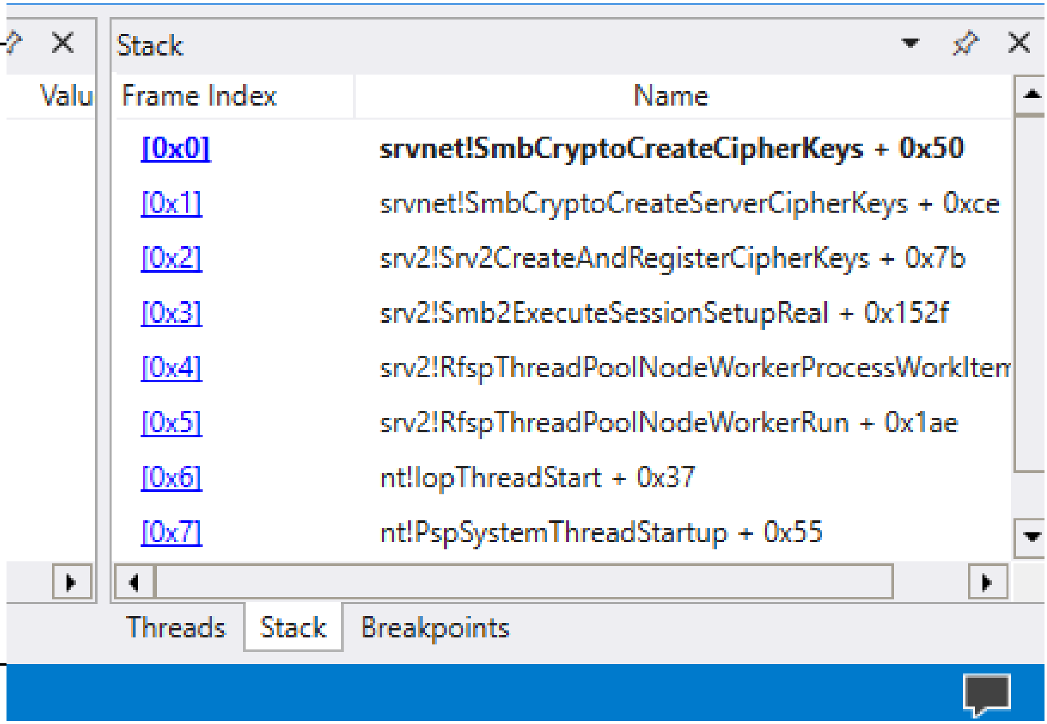


```
Smb2AllocateSession proc near          ; CODE XREF: Smb2ExecuteSessionSetupReal+D8↓p
                                       ; DATA XREF: .rdata:00000001C0034FC4↑o ...

var_28          = qword ptr -28h
arg_0           = qword ptr  8
arg_8           = qword ptr  10h
arg_10          = qword ptr  18h

; FUNCTION CHUNK AT PAGE:00000001C00618F2 SIZE 0000006D BYTES

                mov     [rsp+arg_0], rbx
                mov     [rsp+arg_8], rbp
                mov     [rsp+arg_10], rsi
                push    rdi
                push    r13
                push    r15
                sub     rsp, 30h
                mov     rdi, [rcx+1F0h]
                mov     rsi, rcx
                mov     r15d, 7332534Ch
                mov     edx, 230h       ; NumberOfBytes
                mov     r8d, r15d       ; Tag
                mov     ecx, 1          ; PoolType
                call    cs:__imp_ExAllocatePoolWithTag
                nop     dword ptr [rax+rax+00h]
                xor     ebp, ebp
```

Key generation

- After looking around some more, the code path we are interested in is:

```
srv2.Smb2ExecuteSessionSetupReal() {
    srv2.Srv2CreateAndRegisterCipherKeys() {
        srvnet.SmbCryptoCreateServerCipherKeys() {
            srvnet.SmbCryptoCreateCipherKeys() {
                // key derivation and generation via BCrypt API
                BCryptGenerateSymmetricKey()
                // return opaque BCrypt handles containing the
                // keys
            }
        }
    }
}
```



Frame Index	Name
[0x0]	**srvnet!SmbCryptoCreateCipherKeys + 0x50**
[0x1]	srvnet!SmbCryptoCreateServerCipherKeys + 0xce
[0x2]	srv2!Srv2CreateAndRegisterCipherKeys + 0x7b
[0x3]	srv2!Smb2ExecuteSessionSetupReal + 0x152f
[0x4]	srv2!RfspThreadPoolNodeWorkerProcessWorkItem
[0x5]	srv2!RfspThreadPoolNodeWorkerRun + 0x1ae
[0x6]	nt!IopThreadStart + 0x37
[0x7]	nt!PspSystemThreadStartup + 0x55

Key generation

- BCrypt?
  - Standard, documented, Windows crypto API
  - https://docs.microsoft.com/en-us/windows/win32/api/bcrypt/

```
NTSTATUS BCryptGenerateSymmetricKey(
  BCRYPT_ALG_HANDLE hAlgorithm,
  BCRYPT_KEY_HANDLE *phKey, <====== generated key!
  PUCHAR            pbKeyObject,
  ULONG             cbKeyObject,
  PUCHAR            pbSecret,
  ULONG             cbSecret,
  ULONG             dwFlags
);
```

Key generation

- `BCRYPT_KEY_HANDLE` is an opaque pointer type though...
  - We are looking for an AES-128 key
  - 128 bits = 16 bytes
  - The plan is now to
    - Put a breakpoint in the server after the keys are generated
    - Connect Samba smbclient to the debugged server
    - Dump smbclient client key (via an existing command line argument)
    - In the debugger, inspect the memory of the `BCRYPT_KEY_HANDLE`

Finding the key in the BCrypt handle

- BCRYPT_KEY_HANDLE is a void pointer, we don't know the struct content or size
  - How to tell plain data apart from addresses?
    - Kernel memory lives on the high end of memory
    - All addresses will start 0xfffff....
- Plan is now to inspect memory at the handle, and recursively repeat for things that look like addresses
  - X86_64 systems have 8 bytes addresses
  - Use dq in WinDbg to dump data as 8 bytes ints (will reverse the bytes on little endian)
- Fortunately, the key bytes are found relatively quickly at

```
aeskey = (uint8_t*)(*(uint64*)(*key_handle)) + 92;
```

Finding the key in the BCrypt handle

```
aeskey = (uint8_t*)(*(uint64*)(*key_handle)) + 92;
```

- Essentially:

-
  ```
  struct BCRYPT_KEY {
      struct substruct {
          // 92 bytes of data here
          uint8_t aes128key[16];
      } *ptr;
      // more data here
  };
  ```

Automating key dumping

- Now that we know when and where the key is stored, how can we automate it?
- Solution A: Patching srv2.sys
  - Will fail code signing
  - Tricky to add additional functions imports if we want to use simple file io API
  - Needs to be re-figured out for every build of srv2.sys

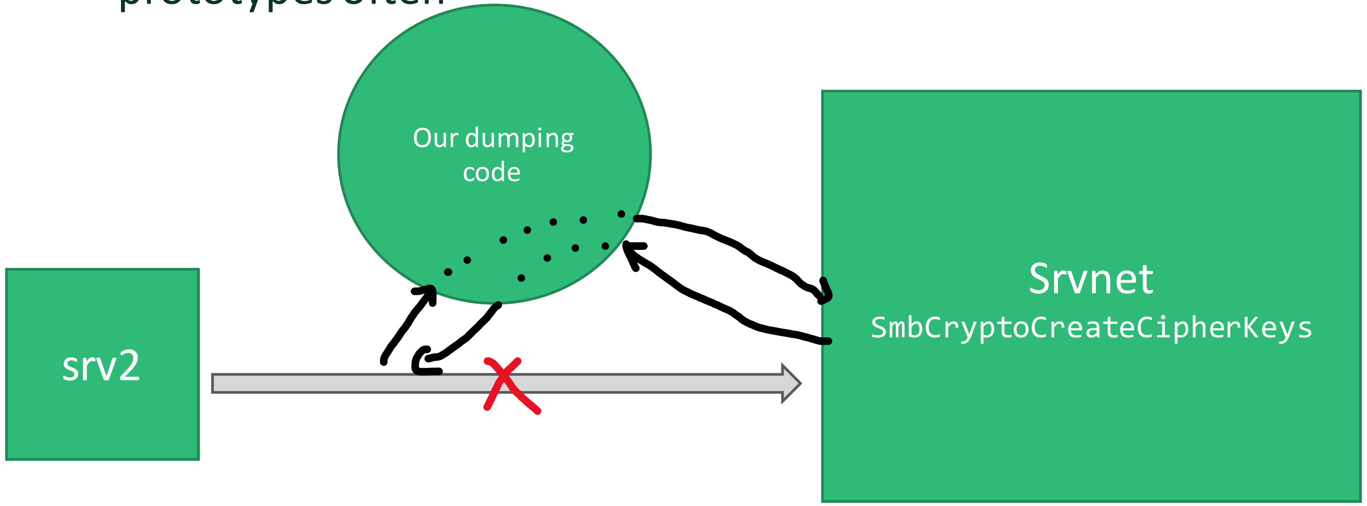
Automating key dumping

- Solution B: Writing a new driver that patches srv2.sys in memory
  - We can self-sign it
  - Using any API is easy
  - We can hook our dumping code at the exact right spot
  - Still need to re-figure offsets and such for every build of srv2.sys

Automating key dumping

- Now that we know when and where the key is stored, how can we automate it?
- Solution C: Writing a new driver that hooks into srv2.sys imports
  - srv2 calls into the srvnet module
  - srvnet exported functions are less likely to change prototypes often

```
srv2.Smb2ExecuteSessionSetupReal() {
    srv2.Srv2CreateAndRegisterCipherKeys() {
        srvnet.SmbCryptoCreateServerCipherKeys() {
            srvnet.SmbCryptoCreateCipherKeys() {
                // key derivation and generation via BCrypt API
                BCryptGenerateSymmetricKey()
                // return opaque BCrypt handles containing the
                // keys
            }
        }
    }
}
```



Refresher on loading

- PE files can export symbols (libs) and import symbols (extern calls)
- The PE header has an Import and Export table section
  (PLT, Procedure Linkage Table)
- Those tables list the symbol name (ascii string) and the address where that function can be called
- Addresses are zeroes on disk, but once loaded in memory, the linker mixes and matches imports with exports from the modules already loaded

Refresher on loading

```
a.sys (about to be loaded)
```

Imports:

- `func_in_b,` **`0x????????`**

Exports:

- (nothing)

Code:

```
call import_table[func_in_b]
```

```
b.sys (loaded at 'base')
```

Imports:

- (nothing)

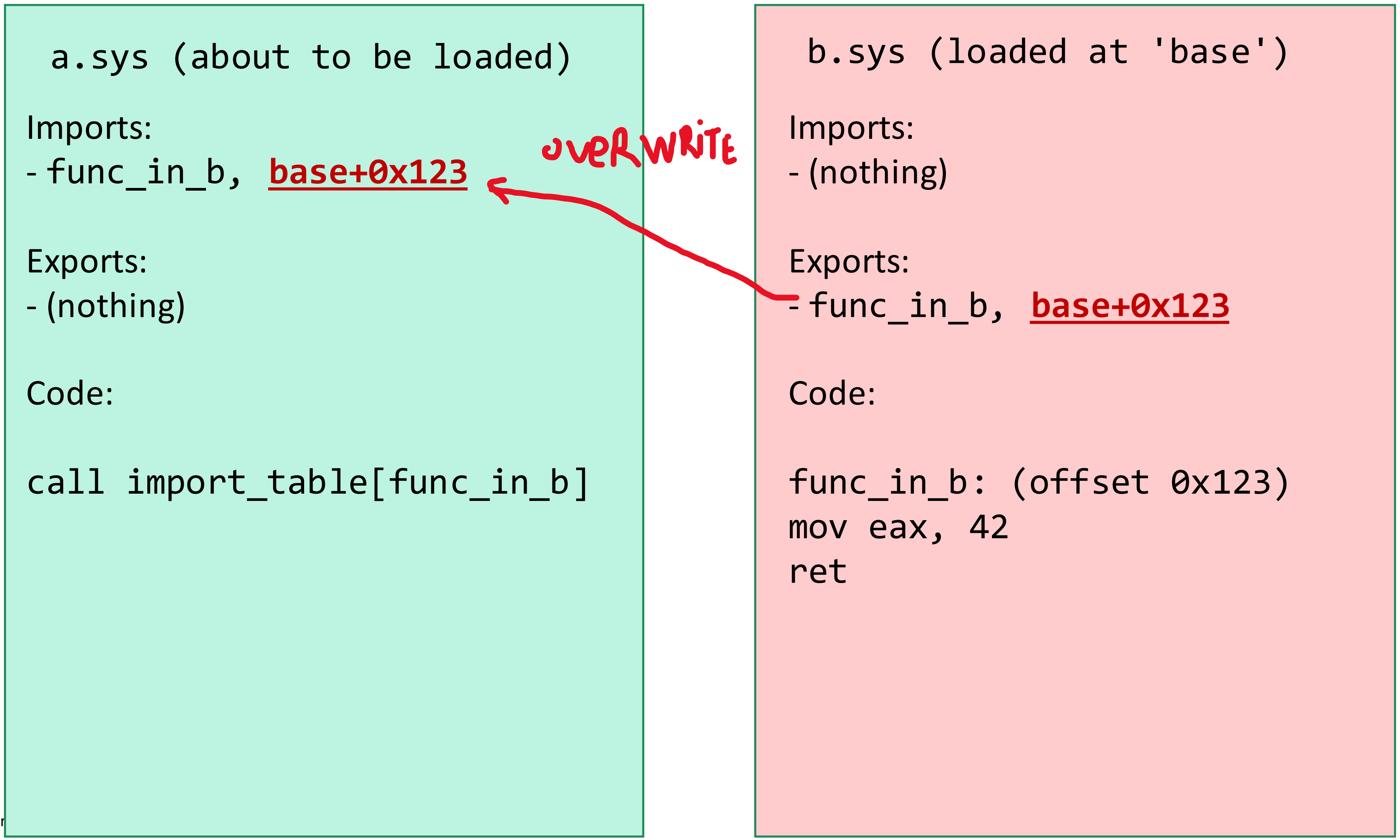
Exports:

- `func_in_b,` **`base+0x123`**

Code:

```
func_in_b: (offset 0x123)
mov eax, 42
ret
```

Refresher on loading



```
a.sys (about to be loaded)
```

Imports:

- `func_in_b, base+0x123`

Exports:

- (nothing)

Code:

```
call import_table[func_in_b]
```

```
b.sys (loaded at 'base')
```

Imports:

- (nothing)

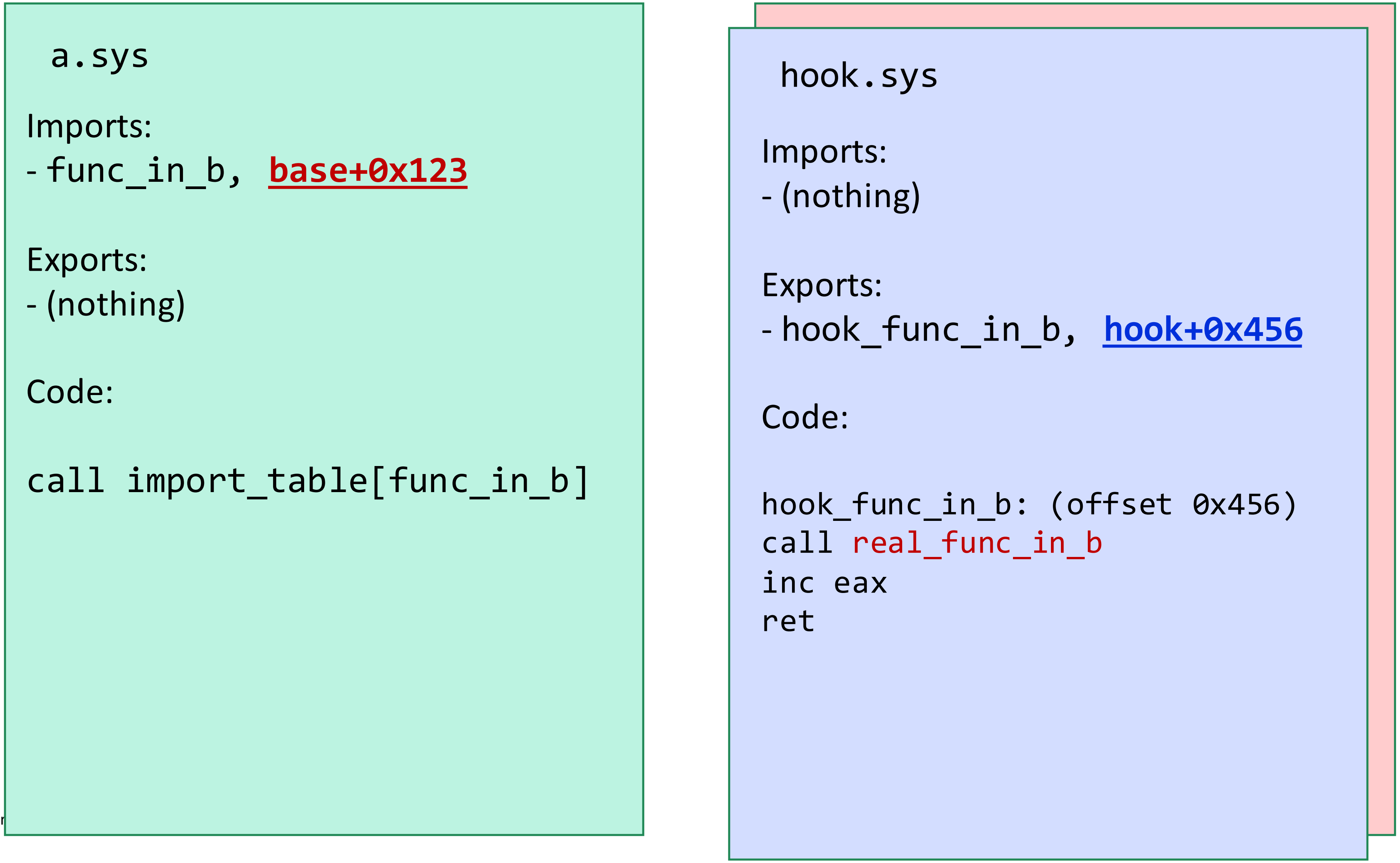
Exports:

- `func_in_b, base+0x123`

Code:

```
func_in_b: (offset 0x123)
mov eax, 42
ret
```

Refresher on loading: hooking



a.sys

Imports:
- func_in_b, **base+0x123**

Exports:
- (nothing)

Code:

```
call import_table[func_in_b]
```

hook.sys

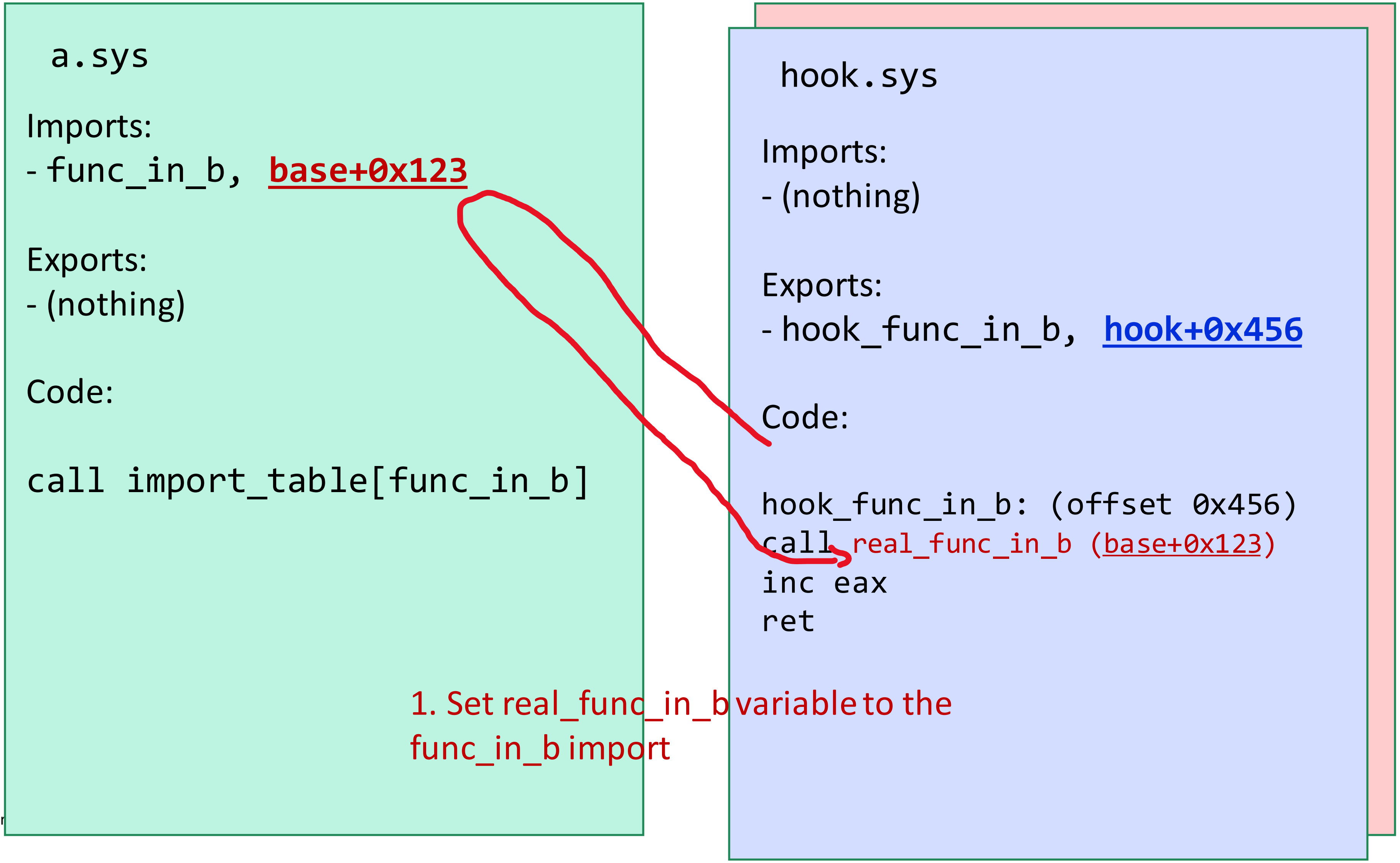
Imports:
- (nothing)

Exports:
- hook_func_in_b, **hook+0x456**

Code:

```
hook_func_in_b: (offset 0x456)
call real_func_in_b
inc eax
ret
```

Refresher on loading: hooking



a.sys

Imports:

- func_in_b, base+0x123

Exports:

- (nothing)

Code:

```
call import_table[func_in_b]
```

hook.sys

Imports:

- (nothing)

Exports:

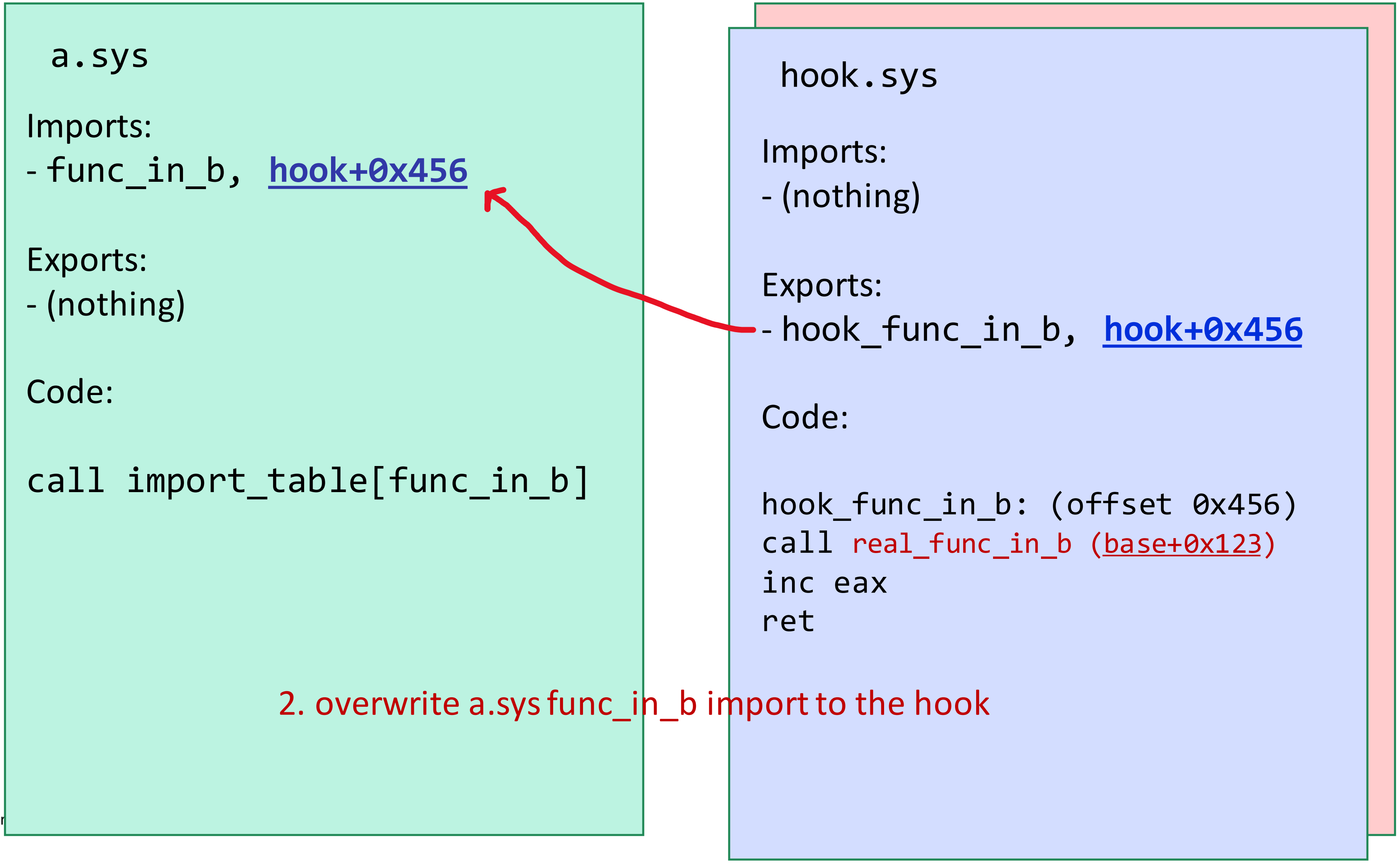
- hook_func_in_b, hook+0x456

Code:

```
hook_func_in_b: (offset 0x456)
call real_func_in_b (base+0x123)
inc eax
ret
```

1. Set real_func_in_b variable to the func_in_b import

Refresher on loading: hooking



```
 a.sys
```

Imports:

- `func_in_b,` `hook+0x456`

Exports:

- (nothing)

Code:

```
call import_table[func_in_b]
```

```
 hook.sys
```

Imports:

- (nothing)

Exports:

- `hook_func_in_b,` `hook+0x456`

Code:

```
hook_func_in_b: (offset 0x456)
call real_func_in_b (base+0x123)
inc eax
ret
```

2. overwrite a.sys func_in_b import to the hook

Refresher on loading: hooking

`a.sys`

Imports:
- `func_in_b, hook+0x456`

Exports:
- (nothing)

Code:

```
call import_table[func_in_b]
```

`hook.sys`

Imports:
- (nothing)

Exports:
- `hook_func_in_b, hook+0x456`

Code:

```
hook_func_in_b: (offset 0x456)
call real_func_in_b (base+0x123)
inc eax
ret
```

`b.sys (loaded at 'base')`

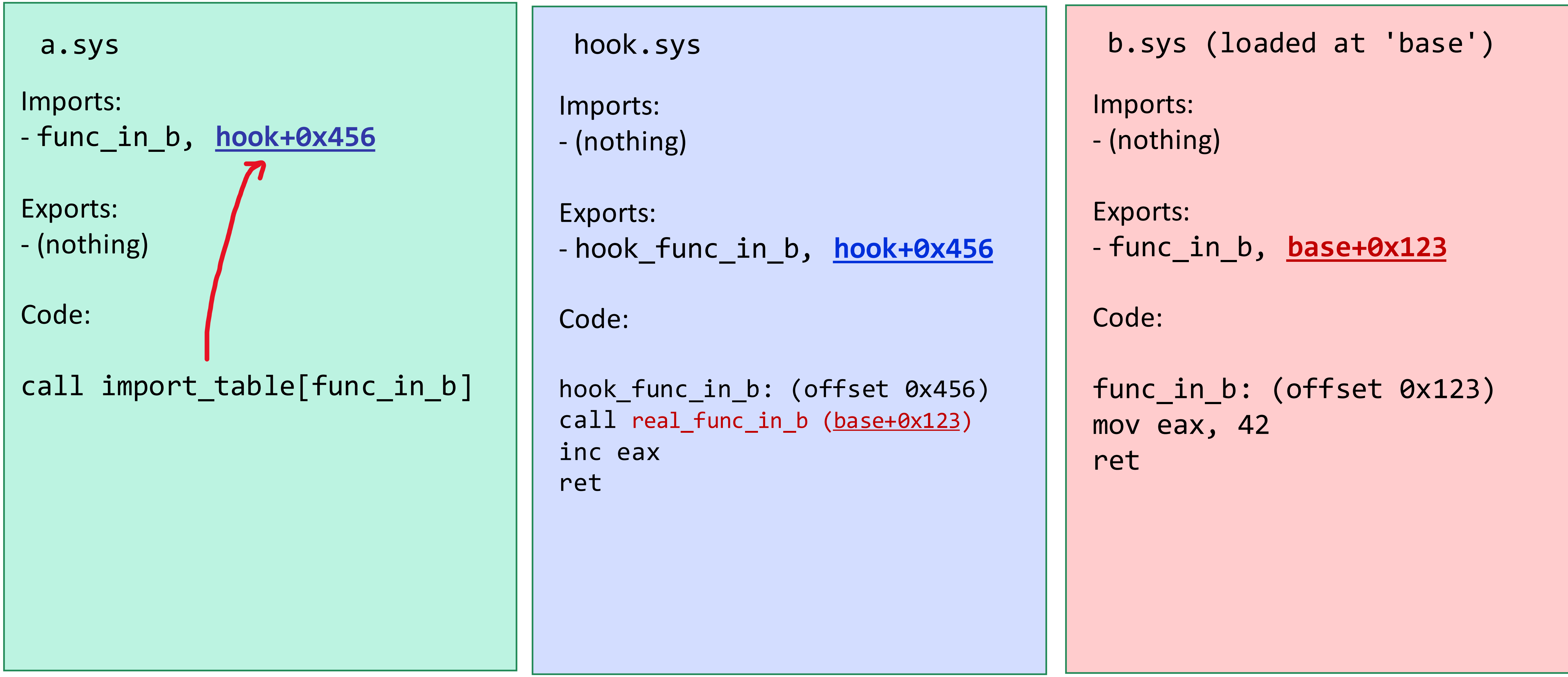
Imports:
- (nothing)

Exports:
- `func_in_b, base+0x123`

Code:

```
func_in_b: (offset 0x123)
mov eax, 42
ret
```

Refresher on loading: hooking



```
a.sys
```

Imports:
- `func_in_b, hook+0x456`

Exports:
- (nothing)

Code:

```
call import_table[func_in_b]
```

```
hook.sys
```

Imports:
- (nothing)

Exports:
- `hook_func_in_b, hook+0x456`

Code:

```
hook_func_in_b: (offset 0x456)
call real_func_in_b (base+0x123)
inc eax
ret
```

```
b.sys (loaded at 'base')
```

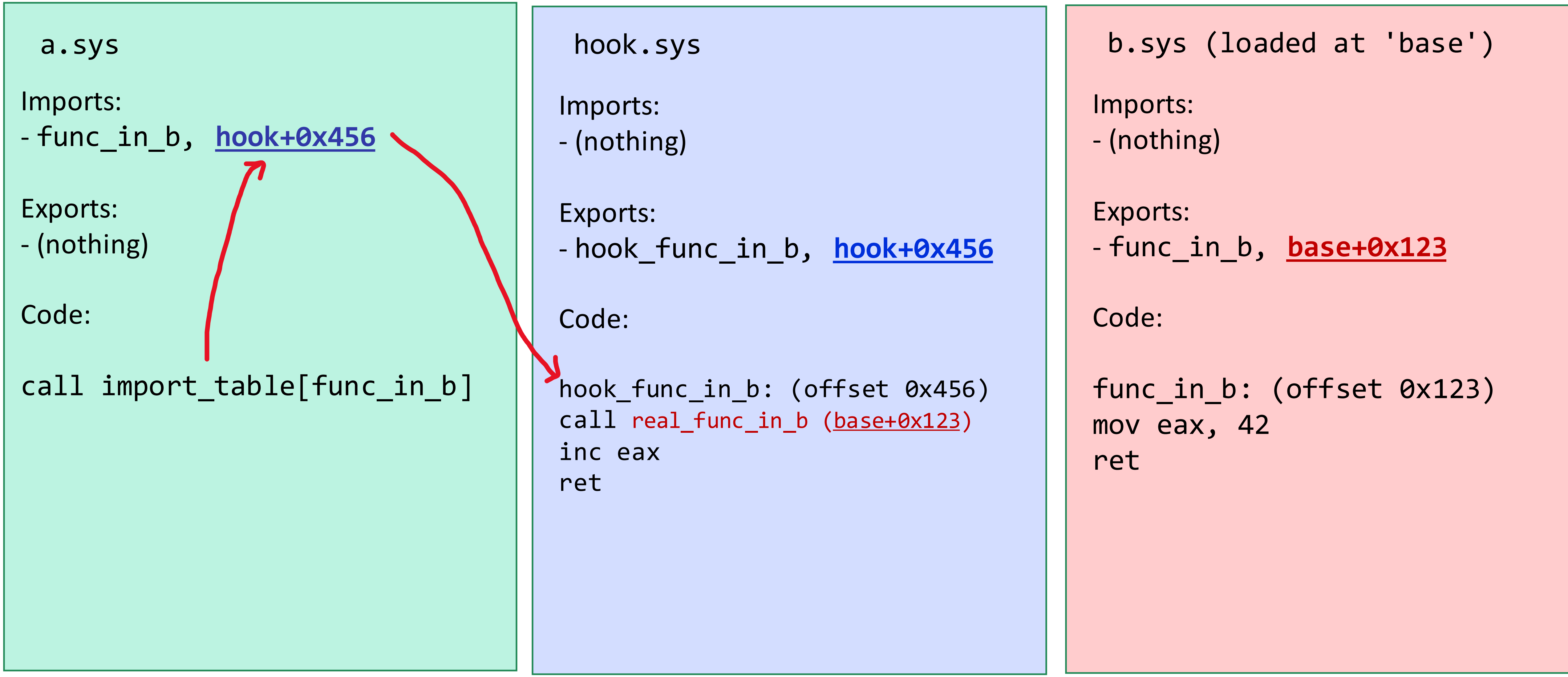
Imports:
- (nothing)

Exports:
- `func_in_b, base+0x123`

Code:

```
func_in_b: (offset 0x123)
mov eax, 42
ret
```

Refresher on loading: hooking



```
a.sys
```

Imports:

- `func_in_b, hook+0x456`

Exports:

- (nothing)

Code:

```
call import_table[func_in_b]
```

```
hook.sys
```

Imports:

- (nothing)

Exports:

- `hook_func_in_b, hook+0x456`

Code:

```
hook_func_in_b: (offset 0x456)
call real_func_in_b (base+0x123)
inc eax
ret
```

```
b.sys (loaded at 'base')
```

Imports:

- (nothing)

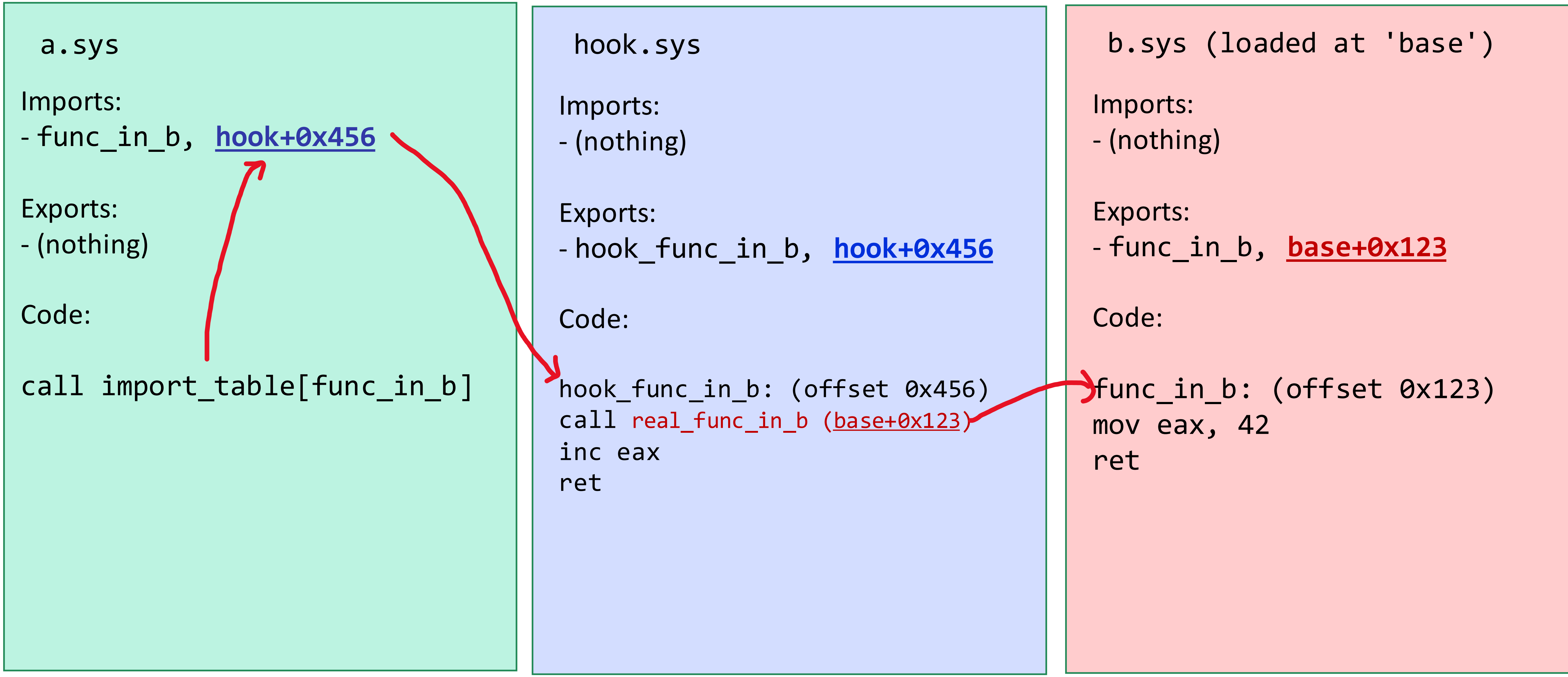
Exports:

- `func_in_b, base+0x123`

Code:

```
func_in_b: (offset 0x123)
mov eax, 42
ret
```

Refresher on loading: hooking



```
a.sys
```

Imports:
- `func_in_b, hook+0x456`

Exports:
- (nothing)

Code:

```
call import_table[func_in_b]
```

```
hook.sys
```

Imports:
- (nothing)

Exports:
- `hook_func_in_b, hook+0x456`

Code:

```
hook_func_in_b: (offset 0x456)
call real_func_in_b (base+0x123)
inc eax
ret
```

```
b.sys (loaded at 'base')
```

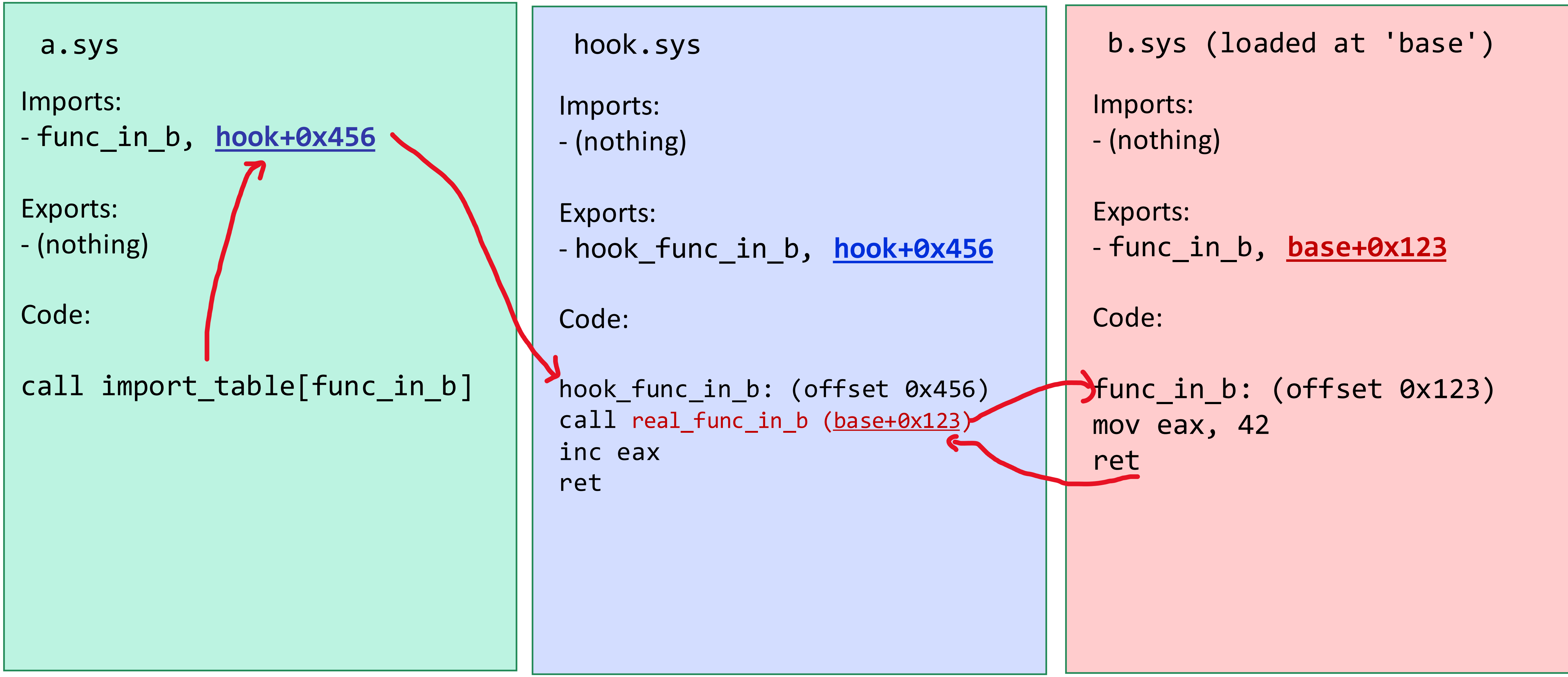
Imports:
- (nothing)

Exports:
- `func_in_b, base+0x123`

Code:

```
func_in_b: (offset 0x123)
mov eax, 42
ret
```

Refresher on loading: hooking



```
a.sys
```

Imports:
- func_in_b, hook+0x456

Exports:
- (nothing)

Code:

```
call import_table[func_in_b]
```

```
hook.sys
```

Imports:
- (nothing)

Exports:
- hook_func_in_b, hook+0x456

Code:

```
hook_func_in_b: (offset 0x456)
call real_func_in_b (base+0x123)
inc eax
ret
```

```
b.sys (loaded at 'base')
```

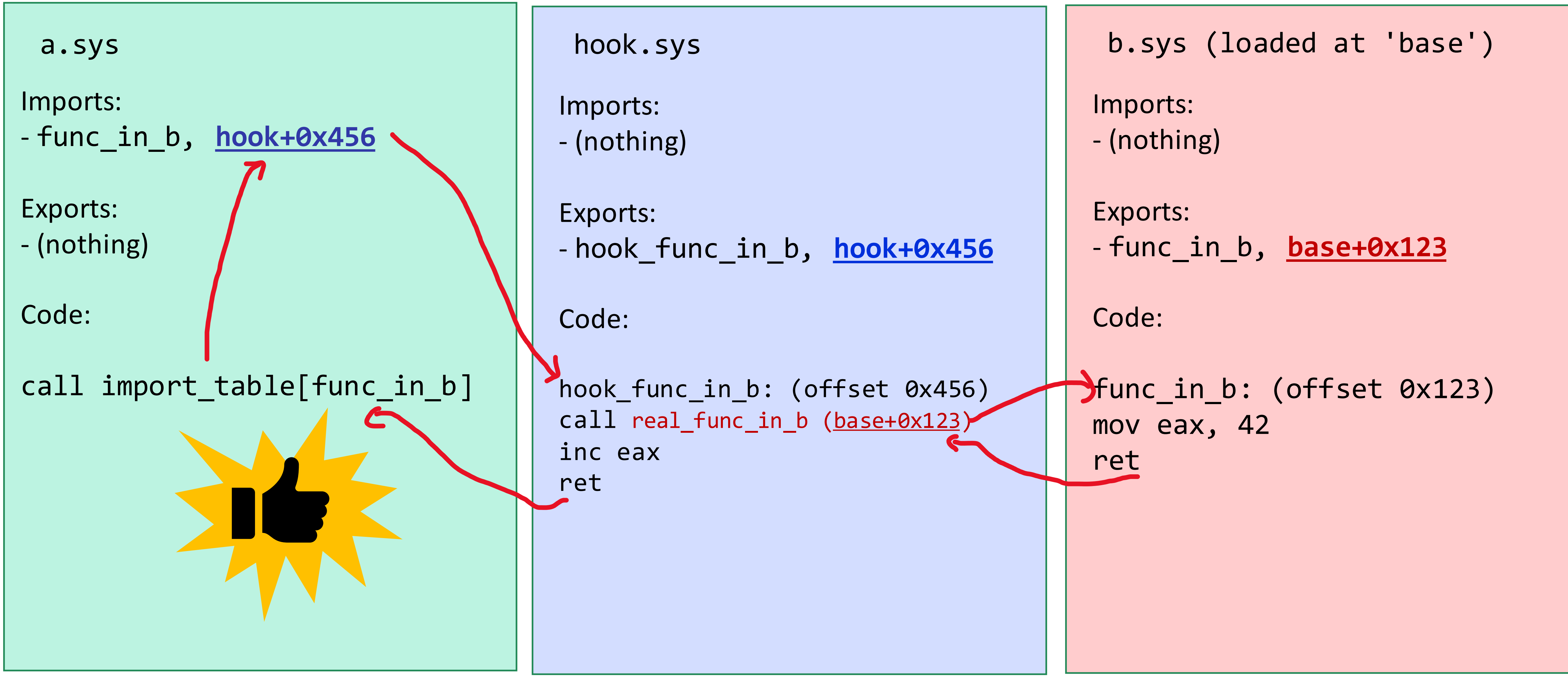
Imports:
- (nothing)

Exports:
- func_in_b, base+0x123

Code:

```
func_in_b: (offset 0x123)
mov eax, 42
ret
```

Refresher on loading: hooking



a.sys

Imports:
- func_in_b, hook+0x456

Exports:
- (nothing)

Code:

```
call import_table[func_in_b]
```

hook.sys

Imports:
- (nothing)

Exports:
- hook_func_in_b, hook+0x456

Code:

```
hook_func_in_b: (offset 0x456)
call real_func_in_b (base+0x123)
inc eax
ret
```

b.sys (loaded at 'base')

Imports:
- (nothing)

Exports:
- func_in_b, base+0x123

Code:

```
func_in_b: (offset 0x123)
mov eax, 42
ret
```

Implementing the driver

- Visual Studio, following Microsoft documentation
- Kernel mode driver
- On load
  - Find srv2.sys module base address in memory
  - Look for `SmbCryptoCreateServerCipherKeys` entry in import table
  - Copy the func address (real func)
  - Overwrite the entry with our function address
- On unload
  - Restore srv2 import table addresses
- Hook function print keys to a file C:\SMBKeyDumpLog.txt

Implementing the driver

- 2 functions needed to be hooked
  - `SmbCryptoCreateServerCipherKeys` to access encryption&decryption keys
  - But also `SmbCryptoKeyTableInsert` to access the Session ID as one of the parameters

Implementing the driver

- Many issues:
  - No API to find module base address
    - Use undocumented call to get the address of kernel module array list
    - Loop over module and look for one with a srv2.sys name attribute
  - Cannot write in read-only memory (import table)
    - Use `MmMapLockedPagesSpecifyCache()` and `MmProtectMdlSystemAddress()` to change read/write permissions on the pages the import table is.
  - Number and size of arguments of hooked functions
    - Windows x64 "fastcall" ABI
    - https://docs.microsoft.com/en-us/cpp/build/x64-calling-convention?view=msvc-160#parameter-passing

Implementing the driver

- Many issues:
  - Tried issuing my own certificate and self-sign but impossible to get it to work
    - Need to boot in signing debug mode to load it
  - Windows doesn't have simple insmod/rmmod to load/unload kernel modules
    - Tried understanding driver .inf file but couldn't figure it out
    - Used OSR Loader
    - Point it at .sys file, click load/unload buttons
    - https://www.osronline.com/article.cfm%5Earticle=157.htm

Live demo

- Dumping & decrypting

Final words, credits, questions

- Code for the driver on github https://github.com/aaptel/SMBKeyDump
  - Only tested with a Win10 VM
- Thanks to people on reddit reverse engineering discord server
- The module list trick
  - http://alter.org.ua/docs/nt_kernel/procaddr/
- VirtualKD source code for changing page mode bits
  - https://github.com/4d61726b/VirtualKD-Redux

SUSE

Thank you

For more information, contact SUSE at:

+1 800 796 3700 (U.S./Canada)

+49 (0)911-740 53-0 (Worldwide)

Maxfeldstrasse 5
90409 Nuremberg
www.suse.com