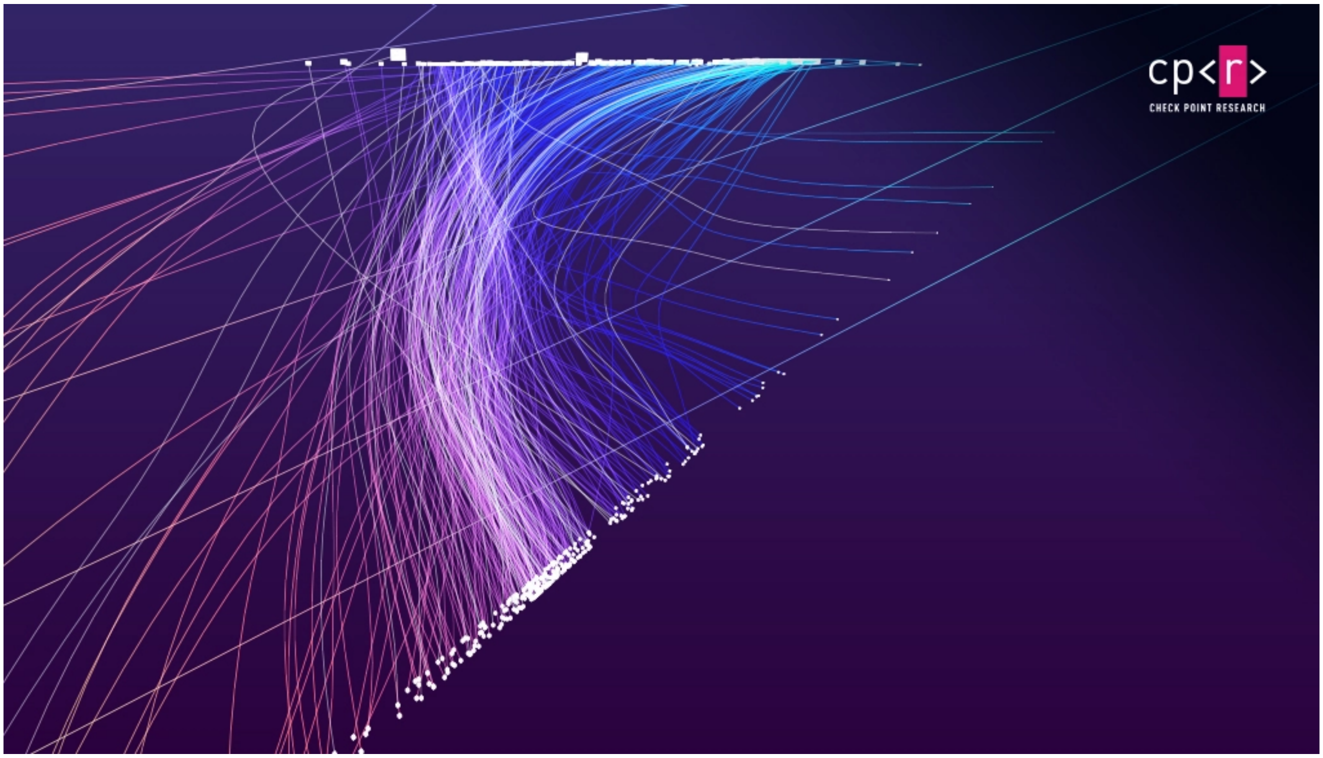
Native function and Assembly Code Invocation

September 21, 2022



September 21, 2022
Author: Jiri Vinopal

Introduction

For a reverse engineer, the ability to directly call a function from the analyzed binary can be a shortcut that bypasses a lot of grief. While in some cases it is just possible to understand the function logic and reimplement it in a higher-level language, this is not always feasible, and it becomes less feasible the more the logic of the original function is fragile and sophisticated. This is an especially sore issue when dealing with custom hashing and encryption — a single off-by-one error somewhere in the computation will cause complete divergence of the final output, and is a mighty chore to debug.

In this article, we walk through 3 different ways to make this "shortcut" happen, and invoke functions directly from assembly. We first cover the IDA Appcall feature which is natively supported by IDA Pro, and can be used directly using IDAPython. We then demonstrate how to achieve the same feat using Dumpulator; and finally, we will show how to get that result using emulation with Unicorn Engine. The practical example used in this article is based on the "tweaked" SHA1 hashing algorithm implemented by a sample of the MiniDuke malware.

Modified SHA1 Hashing algorithm implemented by MiniDuke

The modified SHA1 algorithm in the `MiniDuke` sample is used to create a per-system encryption key for the malware configuration. The buffer to be hashed contains the current computer name concatenated with DWORDs of all interface descriptions, e.g. `'DESKTOP-ROAC4IJ\x00MicrWAN WAN MicrWAN MicrWAN InteWAN InteWAN Inte'`. This function (`SHA1Hash`) uses the same constants as the original SHA1 for both the initial digest and intermediate stages, but produces different outputs.

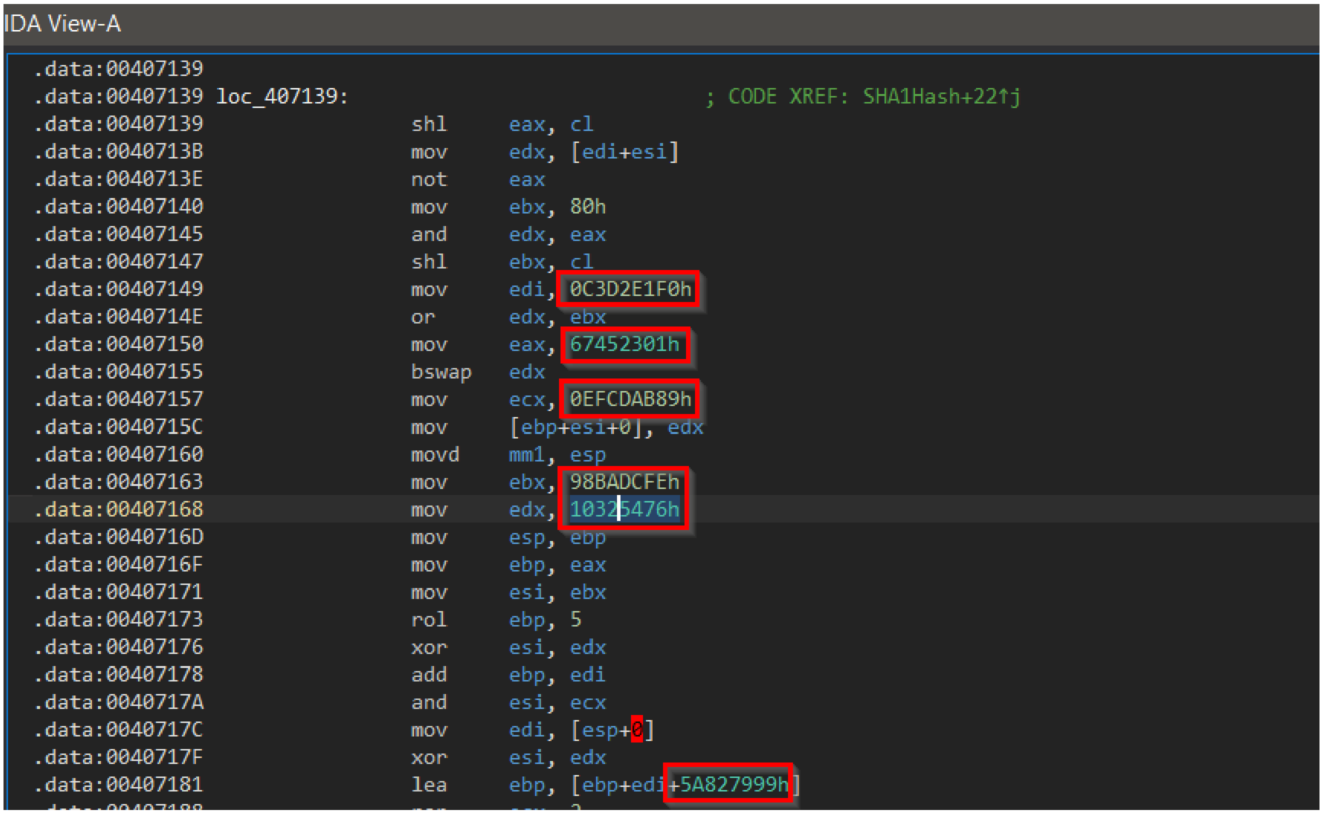


Figure 1: MiniDuke SHA1Hash function constants

Since the constants used are all the same in the original and modified SHA1, the difference must occur somewhere in one of the function's 1,241 assembly instructions. We cannot say whether this tweak was introduced intentionally but the fact remains that malware authors are growing fonder of inserting "surprises" like this, and it falls to analysts to deal with them. To do so, we must first understand in what form the function expects its input and produces its output.

As it turns out, the Duke-SHA1 assembly uses a custom calling convention where the length of buffer to be hashed is passed in the `ecx` register and the address of the buffer itself in `edi` . A value is technically also passed in `eax` but this value is identically `0xffffffff` whenever the executable invokes the function, and we can treat it as a constant for our purposes. Interestingly, the malware also sets the buffer length (`ecx`) to `0x40` every time it invokes this function, effectively hashing only the first `0x40` bytes of the buffer.

```
mov     ecx, 40h ; '@'  ; buffLen
xor     eax, eax
dec     eax             ; int
lea     edi, [ebp+IFandPcNameBuff] ; buffer
call    SHA1Hash        ; compute SHA1 from PCNameIF buffer
```

Figure 2: SHA1Hash function arguments

The resulting 160-bit SHA1 hash value is returned in 5 dwords in registers (from high dword to low: `eax` , `edx` , `ebx` , `ecx` , `esi`). For example, the buffer `DESKTOP-ROAC4IJ\x00MicrWAN WAN MicrWAN MicrWAN InteWAN InteWAN Inte` has a Duke-SHA1 value of `1851fff77f0957d1d690a32f31df2c32a1a84af7` , returned as `EAX:0x1851fff7 EDX:0x7f0957d1 EBX:0xd690a32f ECX:0x31df2c32 ESI:0xa1a84af7` .

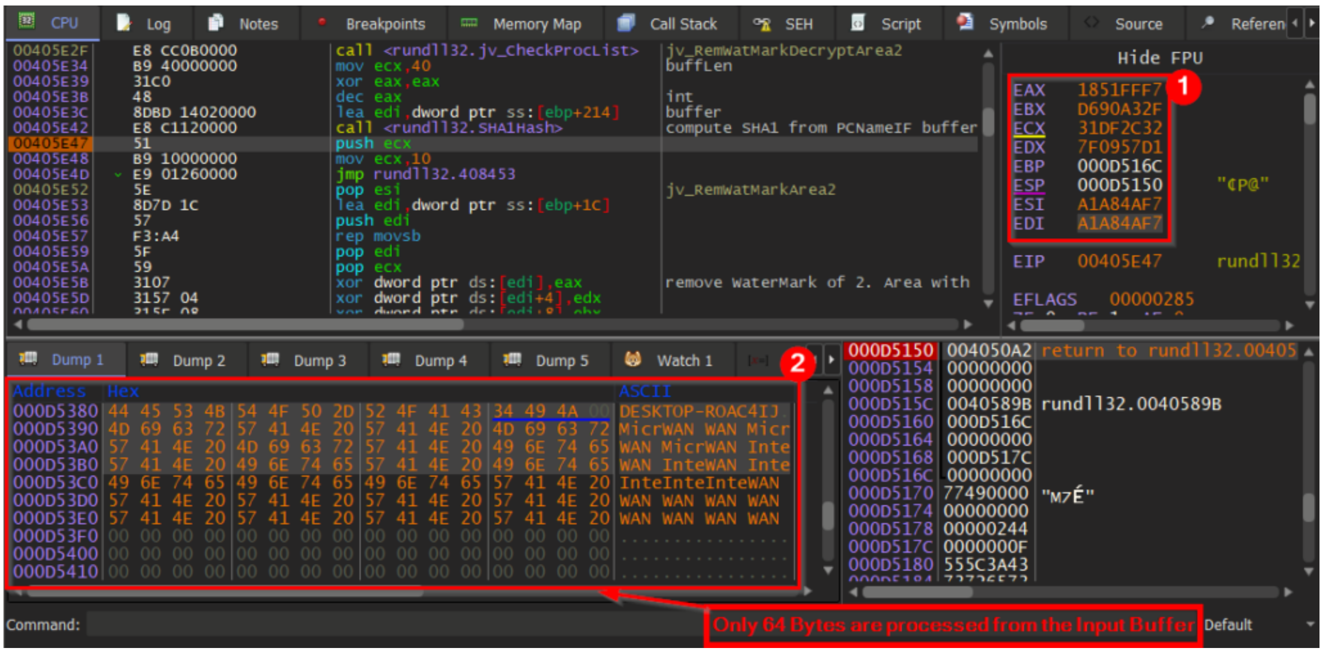


Figure 3: Example produced SHA1 Hash of buffer

As explained before, hunting down the exact place(s) where the logic of SHA1 and Duke-SHA1 diverge and then reimplementing Duke-SHA1 in Python is an excellent way to waste a day, and possibly a week. Instead, we will use several approaches to “plug into” the function’s calling convention and invoke it directly.

IDA – Appcall

`Appcall` is a feature of IDA Pro which allows IDA Python scripts to call functions inside the debugged program as if they were built-in functions. This is very convenient, but it also suffers from the typical curse of convenient solutions, which is a very sharp spike in difficulty of application when the use case gets somewhat unusual or complex. Alas, such is the case here: while passing a buffer length in `ecx` and a buffer in `edi` is par for the course, the 160-bit return value split across 5 registers is not your typical form of function output, and Appcall requires some creative coercion to cooperate with what we want it to do here.

We proceed by creating a custom structure `struc_SHA1HASH` which holds the values of 5 registers, and is used as a return type of the function prototype:

```
STRUCT_NAME = "struc_SHA1HASH"
# ------------------Struct Creation ------------------
sid = idc.get_struc_id(STRUCT_NAME)
if (sid != -1):
    idc.del_struc(sid)
sid = idc.add_struc(-1, STRUCT_NAME, 0)
idc.add_struc_member(sid, "_EAX_", -1, idc.FF_DWORD, -1, 4)
idc.add_struc_member(sid, "_EDX_", -1, idc.FF_DWORD, -1, 4)
idc.add_struc_member(sid, "_EBX_", -1, idc.FF_DWORD, -1, 4)
idc.add_struc_member(sid, "_ECX_", -1, idc.FF_DWORD, -1, 4)
idc.add_struc_member(sid, "_ESI_", -1, idc.FF_DWORD, -1, 4)
```

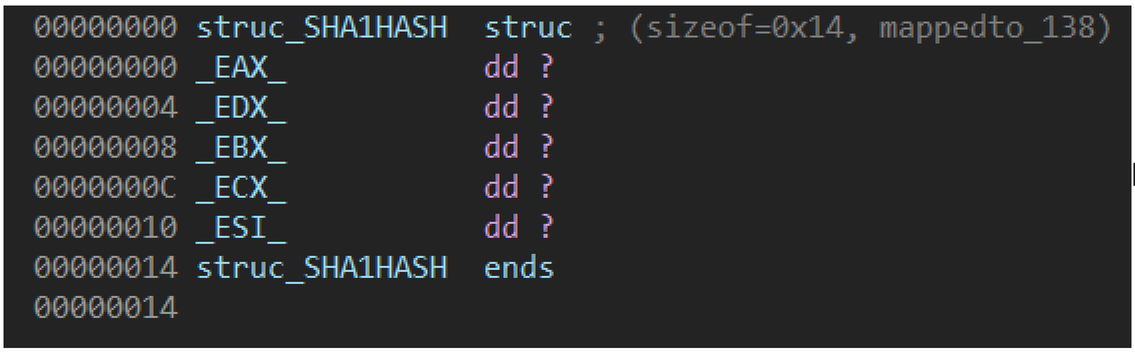


Figure 4: IDA Structure Window – “struc_SHA1HASH”

Now with the structure definition in place, we are poised to invoke the magic incantation that will allow Appcall to interface with this function prototype, as seen in the `PROTO` value below.

```
# ------------------Initialization ------------------
FUNC_NAME = "SHA1Hash"
STRUCT_NAME = "struc_SHA1HASH"
PROTO = "{:s} __usercall {:s}@<0:eax, 4:edx, 8:ebx, 12:ecx, 16:esi>(int
[email protected]<ecx>, const [email protected]<eax>, BYTE *[email protected]
<edi>);".format(STRUCT_NAME, FUNC_NAME) # specify prototype of SHA1Hash function
SHA1BUFF_LEN = 0x40
CONSTVAL = 0xffffffff
```

As IDA Appcall relies on the debugger, to invoke this logic we first need to write a script that will start the debugger, make required adjustments to the stack and do other required housekeeping.

```
# ------------------ Setting + Starting Debugger ------------------
idc.load_debugger("win32",0)                 # select Local Windows Debugger
idc.set_debugger_options(idc.DOPT_ENTRY_BPT) # break on program entry point
idc.start_process("","","")                  # start process with default options
idc.wait_for_next_event(idc.WFNE_SUSP, 3)    # waits until process get suspended on
entrypoint
eip = idc.get_reg_value("eip")               # get EIP
idc.run_to(eip + 0x1d)                       # let the stack adjust itself (execute
few instructions)
idc.wait_for_next_event(idc.WFNE_SUSP, 3)    # waits until process get suspended
after stack adjustment
```

Figure 5: IDA View – Stack adjusting

Using Appcall is the last step, and there are several ways to utilize it to call functions. We can call the function directly without specifying a prototype, but this highly relies on a properly typed function in IDA's IDB. The second way is to create a callable object from the function name and a defined prototype. This way we can call a function with a specific prototype, no matter what type is set in the IDB, as shown below:

```
SHA1Hash = Appcall.proto(FUNC_NAME, PROTO)   # creating callable object
inBuff = Appcall.byref(b'DESKTOP-ROAC4IJ\x00MicrWAN WAN MicrWAN MicrWAN InteWAN
InteWAN Inte')
buffLen = SHA1BUFF_LEN
const = CONSTVAL

retValue = SHA1Hash(buffLen, const, inBuff)
eax = malduck.DWORD(retValue._EAX_)
edx = malduck.DWORD(retValue._EDX_)
ebx = malduck.DWORD(retValue._EBX_)
ecx = malduck.DWORD(retValue._ECX_)
esi = malduck.DWORD(retValue._ESI_)
```

The full script to call Duke-SHA1 using Appcall is reproduced below.

```
# IDAPython script to demonstrate Appcall feature on modified SHA1 Hashing algorithm
implemented by MiniDuke malware sample
# SHA1 HASH is stored in EAX, EDX, EBX, ECX, ESI (return values)
# SHA1 HASH Arguments -> ECX = 0x40 (buffLen), EAX = 0xFFFFFFFF (const), EDI =  BYTE
*buffer (buffer)

import idc, malduck
from idaapi import Appcall

# ------------------Initialization ------------------
FUNC_NAME = "SHA1Hash"
STRUCT_NAME = "struc_SHA1HASH"
PROTO = "{:s} __usercall {:s}@<0:eax, 4:edx, 8:ebx, 12:ecx, 16:esi>(int
[email protected]<ecx>, const [email protected]<eax>, BYTE *[email protected]
<edi>);".format(STRUCT_NAME, FUNC_NAME) # specify prototype of SHA1Hash function
SHA1BUFF_LEN = 0x40
CONSTVAL = 0xffffffff

# ------------------Struct Creation ------------------
sid = idc.get_struc_id(STRUCT_NAME)
if (sid != -1):
    idc.del_struc(sid)
sid = idc.add_struc(-1, STRUCT_NAME, 0)
idc.add_struc_member(sid, "_EAX_", -1, idc.FF_DWORD, -1, 4)
idc.add_struc_member(sid, "_EDX_", -1, idc.FF_DWORD, -1, 4)
idc.add_struc_member(sid, "_EBX_", -1, idc.FF_DWORD, -1, 4)
idc.add_struc_member(sid, "_ECX_", -1, idc.FF_DWORD, -1, 4)
idc.add_struc_member(sid, "_ESI_", -1, idc.FF_DWORD, -1, 4)

# ------------------ Setting + Starting Debugger ------------------
idc.load_debugger("win32",0)                # select Local Windows Debugger
idc.set_debugger_options(idc.DOPT_ENTRY_BPT) # break on program entry point
idc.start_process("","","")                 # start process with default options
idc.wait_for_next_event(idc.WFNE_SUSP, 3)    # waits until process get suspended on
entrypoint
eip = idc.get_reg_value("eip")              # get EIP
idc.run_to(eip + 0x1d)                      # let the stack adjust itself (execute
few instructions)
idc.wait_for_next_event(idc.WFNE_SUSP, 3)    # waits until process get suspended
after stack adjustment

# ------------------ Arguments + Execution ------------------
SHA1Hash = Appcall.proto(FUNC_NAME, PROTO)   # creating callable object
inBuff = Appcall.byref(b'DESKTOP-ROAC4IJ\x00MicrWAN WAN MicrWAN MicrWAN InteWAN
InteWAN Inte')
buffLen = SHA1BUFF_LEN
const = CONSTVAL

retValue = SHA1Hash(buffLen, const, inBuff)
eax = malduck.DWORD(retValue._EAX_)
edx = malduck.DWORD(retValue._EDX_)
ebx = malduck.DWORD(retValue._EBX_)
```

```
ecx = malduck.DWORD(retValue._ECX_)
esi = malduck.DWORD(retValue._ESI_)

# ------------------ RESULTS ------------------
print("SHA1 HASH RET VALUES: EAX:0x%x EDX:0x%x EBX:0x%x ECX:0x%x ESI:0x%x" % (eax,
edx, ebx, ecx, esi))

# ------------------ Exiting Debugger ------------------
idc.exit_process()
```

And some sample output:

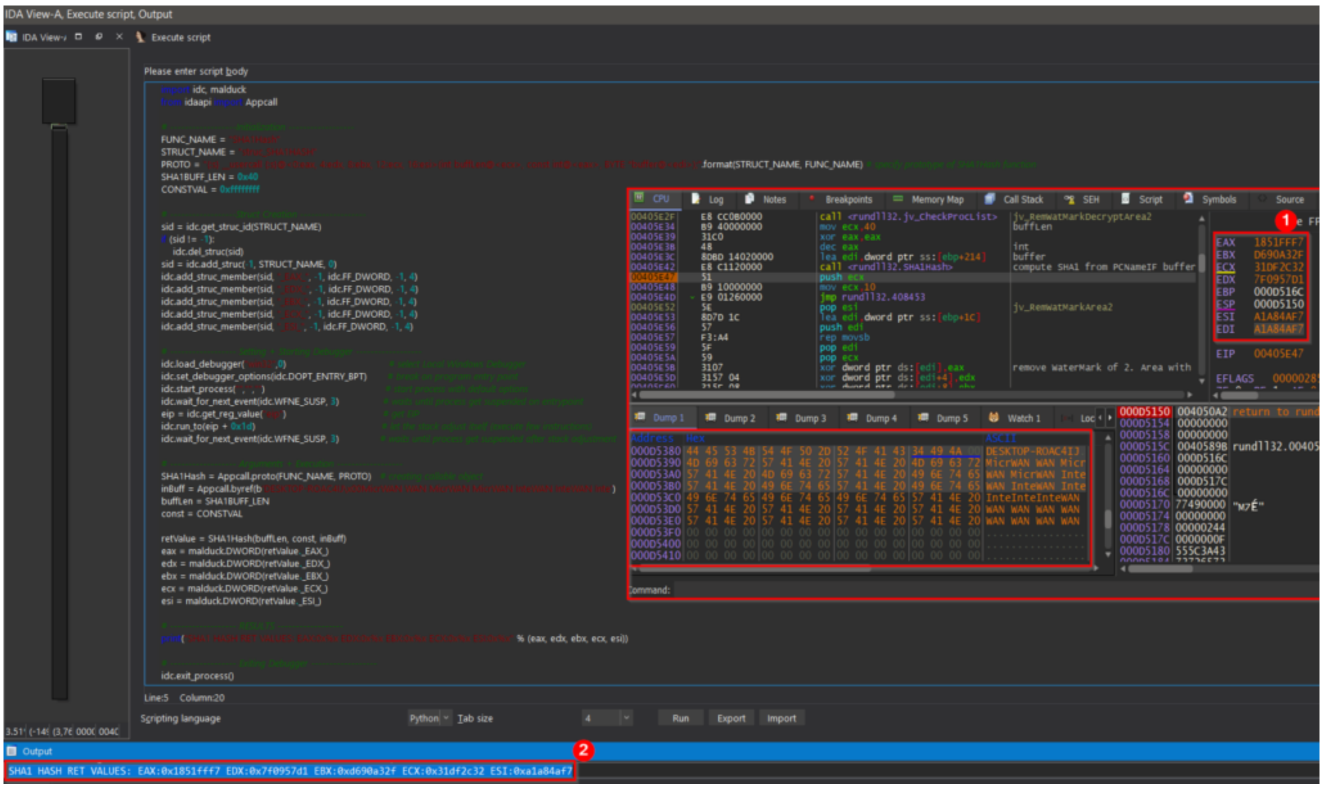


Figure 6: Script execution – “IDA Appcall” producing the same SHA1 Hash values as the MiniDuke sample

The above is fine if we just want to use the invoked function as a black box, but sometimes we may want access to registry values in a specific state of execution, and specifying the prototype as above is something of a chore. Happily, both these downsides can be mitigated, as we will see below.

As IDA Appcall relies on the debugger and can be invoked right from IDAPython, we can invoke Appcall from the debugger and gain more granular control over its execution. For example, we can make Appcall hand control back to the debugger during execution by setting a special option for Appcall – `APPCALL_MANUAL` .

```
# ------------------ Arguments + Execution ------------------
SHA1Hash = Appcall.proto(FUNC_NAME, PROTO)   # creating callable object
SHA1Hash.options = Appcall.APPCALL_MANUAL    # APPCALL_MANUAL option will cause the
debugger to break on function entry and gives the control to debugger
inBuff = Appcall.byref(b'DESKTOP-ROAC4IJ\x00MicrWAN WAN MicrWAN MicrWAN InteWAN
InteWAN Inte')
buffLen = SHA1BUFF_LEN
const = CONSTVAL

SHA1Hash(buffLen, const, inBuff)             # invoking Appcall and breaking on
function entry (SHA1Hash)
```

This way we can make use of Appcall to prepare arguments, allocate a buffer and later restore the previous execution context. We can also avoid specifying the structure type for the return value (type it as `void`) as this will be handled by the debugger. There are more ways to get the return values of the function, so as we are now controlling the debugger, we can use (for example) a conditional breakpoint to print desired values in a specific state of execution (such as on return).

```
# ------------------Set conditional BP on Return ------------------
def SetCondBPonRet():
    cond = """import idc
print("SHA1 HASH RET VALUES: EAX:0x%x EDX:0x%x EBX:0x%x ECX:0x%x ESI:0x%x" %
(idc.get_reg_value("eax"), idc.get_reg_value("edx"), idc.get_reg_value("ebx"),
idc.get_reg_value("ecx"), idc.get_reg_value("esi")))
return True
"""
    func = idaapi.get_func(idc.get_name_ea_simple(FUNC_NAME))
    bpt = idaapi.bpt_t()
    bpt.ea = idc.prev_head(func.end_ea)      # last instruction in function -> should
be return
    bpt.enabled = True
    bpt.type = idc.BPT_SOFT
    bpt.elang = 'Python'
    bpt.condition = cond                     # with script code in condition we can
get or log any values we want
    idc.add_bpt(bpt)
    return bpt                               # return breakpoint object -> will be
deleted later on
```

We can restore the previous state (before Appcall invocation) at any desired moment of execution by calling `cleanup_appcall()`. So in our case, right after hitting the conditional breakpoint.

```
SHA1Hash(buffLen, const, inBuff)            # invoking Appcall and breaking on
function entry (SHA1Hash)
idc.wait_for_next_event(idc.WFNE_SUSP, 3)
idaapi.continue_process()                   # debugger has control now so continue
to hit the new conditional breakpoint
idc.wait_for_next_event(idc.WFNE_SUSP, 3)
idc.del_bpt(bpt.ea)                         # deleting the previously created
conditional breakpoint
Appcall.cleanup_appcall()                   # clean Appcall after hitting the
conditional breakpoint -> return
```

The full script is reproduced below.

```
# IDAPython script to demonstrate Appcall feature on modified SHA1 Hashing algorithm
implemented by MiniDuke malware sample
# SHA1 HASH is stored in EAX, EDX, EBX, ECX, ESI (return values)
# SHA1 HASH Arguments -> ECX = 0x40 (buffLen), EAX = 0xFFFFFFFF (const), EDI =  BYTE
*buffer (buffer)

import idc, idaapi
from idaapi import Appcall

# ------------------ Initialization ------------------
FUNC_NAME = "SHA1Hash"
PROTO = "void __usercall {:s}(int [email protected]<ecx>, const [email protected]
<eax>, BYTE *[email protected]<edi>);".format(FUNC_NAME) # specify prototype of
SHA1Hash function
SHA1BUFF_LEN = 0x40
CONSTVAL = 0xffffffff

# ------------------Set conditional BP on Return ------------------
def SetCondBPonRet():
    cond = """import idc
print("SHA1 HASH RET VALUES: EAX:0x%x EDX:0x%x EBX:0x%x ECX:0x%x ESI:0x%x" %
(idc.get_reg_value("eax"), idc.get_reg_value("edx"), idc.get_reg_value("ebx"),
idc.get_reg_value("ecx"), idc.get_reg_value("esi")))
return True
"""
    func = idaapi.get_func(idc.get_name_ea_simple(FUNC_NAME))
    bpt = idaapi.bpt_t()
    bpt.ea = idc.prev_head(func.end_ea)      # last instruction in function -> should
be return
    bpt.enabled = True
    bpt.type = idc.BPT_SOFT
    bpt.elang = 'Python'
    bpt.condition = cond                     # with script code in condition we can
get or log any values we want
    idc.add_bpt(bpt)
    return bpt                               # return breakpoint object -> will be
deleted later on

# ------------------ Setting + Starting Debugger ------------------
idc.load_debugger("win32",0)                 # select Local Windows Debugger
idc.set_debugger_options(idc.DOPT_ENTRY_BPT) # break on program entry point
bpt = SetCondBPonRet()                       # setting the conditional breakpoint on
function return
idc.start_process("","","")                  # start process with default options
idc.wait_for_next_event(idc.WFNE_SUSP, 3)    # waits until process get suspended on
entrypoint
eip = idc.get_reg_value("eip")               # get EIP
idc.run_to(eip + 0x1d)                       # let the stack adjust itself (execute
few instructions)
idc.wait_for_next_event(idc.WFNE_SUSP, 3)    # waits until process get suspended
after stack adjustment
```

```
# ------------------ Arguments + Execution ------------------
SHA1Hash = Appcall.proto(FUNC_NAME, PROTO)   # creating callable object
SHA1Hash.options = Appcall.APPCALL_MANUAL    # APPCALL_MANUAL option will cause the
debugger to break on function entry and gives the control to debugger
inBuff = Appcall.byref(b'DESKTOP-ROAC4IJ\x00MicrWAN WAN MicrWAN MicrWAN InteWAN
InteWAN Inte')
buffLen = SHA1BUFF_LEN
const = CONSTVAL

SHA1Hash(buffLen, const, inBuff)             # invoking Appcall and breaking on
function entry (SHA1Hash)
idc.wait_for_next_event(idc.WFNE_SUSP, 3)
idaapi.continue_process()                    # debugger has control now so continue
to hit the new conditional breakpoint
idc.wait_for_next_event(idc.WFNE_SUSP, 3)
idc.del_bpt(bpt.ea)                          # deleting the previously created
conditional breakpoint
Appcall.cleanup_appcall()                    # clean Appcall after hitting the
conditional breakpoint -> return

# ------------------ Exiting Debugger ------------------
idc.exit_process()
```

Dumpulator

Dumpulator is a python library that assists with code emulation in minidump files. The core emulation engine of dumpulator is based on Unicorn Engine, but a relatively unique feature among other similar tools is that the entire process memory is available. This brings a performance improvement (emulating large parts of analyzed binary without leaving Unicorn), as well as making life more convenient if we can time the memory dump to when the program context (stack, etc) required to call the function is already in place. Additionally, only syscalls have to be emulated to provide a realistic Windows environment (since everything actually is a legitimate process environment).

A minidump of the desired process could be captured with many tools (x64dbg – MiniDumpPlugin, Process Explorer, Process Hacker, Task Manager) or with the Windows API (`MiniDumpWriteDump`). We can use the x64dbg – MiniDumpPlugin to create a minidump in a state where almost all in the process is already set for SHA1 Hash creation, right before the `SHA1Hash` function call. Note that timing the dump this way is not *necessary*, as the environment can be set up manually in dumpulator after taking the dump; it is just *convenient*.

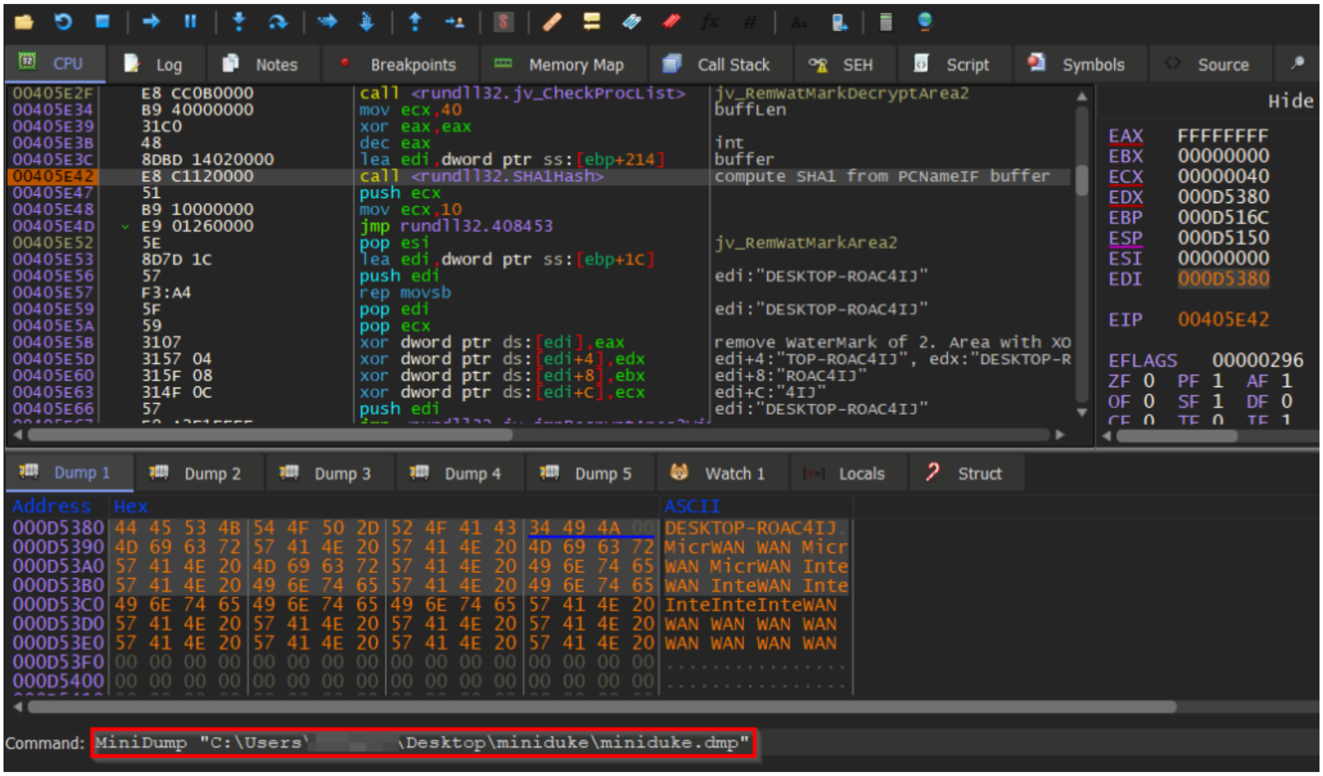


Figure 7: Creation of minidump using “x64dbg – MiniDumpPlugin”

Dumpulator not only has access to the entire dumped process memory but can also allocate additional memory, read memory, write to memory, read registry values, and write registry values. In other words, anything that an emulator can do. There is also a possibility to implement system calls so code using them can be emulated.

To invoke Duke-SHA1 via Dumpulator, we need to specify the address of the function which will be called in `minidump` and its arguments. In this case, the address of `SHA1Hash` is `0x407108` .

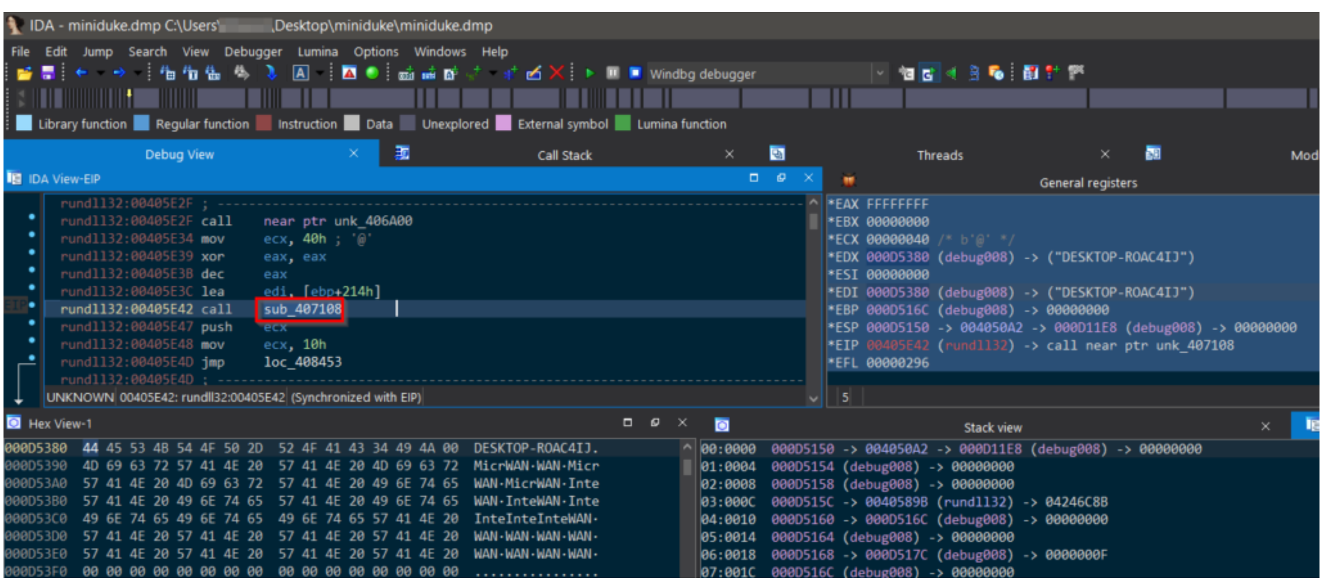


Figure 8: Opening produced minidump in IDA

As we do not want to use already set values in the current state of minidump, we define our own argument values for the function. We can even allocate a new buffer which will be used as a buffer to be hashed. The decidedly elegant code to do this is shown below.

```
# Python script to demonstrate dumpulator on modified SHA1 Hashing algorithm
implemented by MiniDuke malware sample
# SHA1 HASH is stored in EAX, EDX, EBX, ECX, ESI (return values)
# SHA1 HASH Arguments -> ECX = 0x40 (buffLen), EAX = 0xFFFFFFFF (const), EDI =  BYTE
*buffer (buffer)

from dumpulator import Dumpulator

# ------------------Initialization ------------------
FUNC_ADDR = 0x407108            # address of SHA1Hash function in MiniDuke
SHA1BUFF_LEN = 0x40
CONSTVAL = 0xffffffff

# ------------------ Setting + Starting Dumpulator ------------------
dp = Dumpulator("miniduke.dmp", quiet=True)
inBuff = b'DESKTOP-ROAC4IJ\x00MicrWAN WAN MicrWAN MicrWAN InteWAN InteWAN Inte'
bufferAddr = dp.allocate(64)
dp.write(bufferAddr, inBuff)
#dp.regs.ecx = SHA1BUFF_LEN     # possible to set the registers here
#dp.regs.eax = CONSTVAL
#dp.regs.edi = bufferAddr
#dp.call(FUNC_ADDR)
dp.call(FUNC_ADDR, regs= {"eax": CONSTVAL, "ecx": SHA1BUFF_LEN, "edi": bufferAddr})

# ------------------ RESULTS ------------------
print("SHA1 HASH RET VALUES: EAX:0x%x EDX:0x%x EBX:0x%x ECX:0x%x ESI:0x%x" %
(dp.regs.eax, dp.regs.edx, dp.regs.ebx, dp.regs.ecx, dp.regs.esi))
```

Execution of this script will produce correct Duke-SHA1 values.

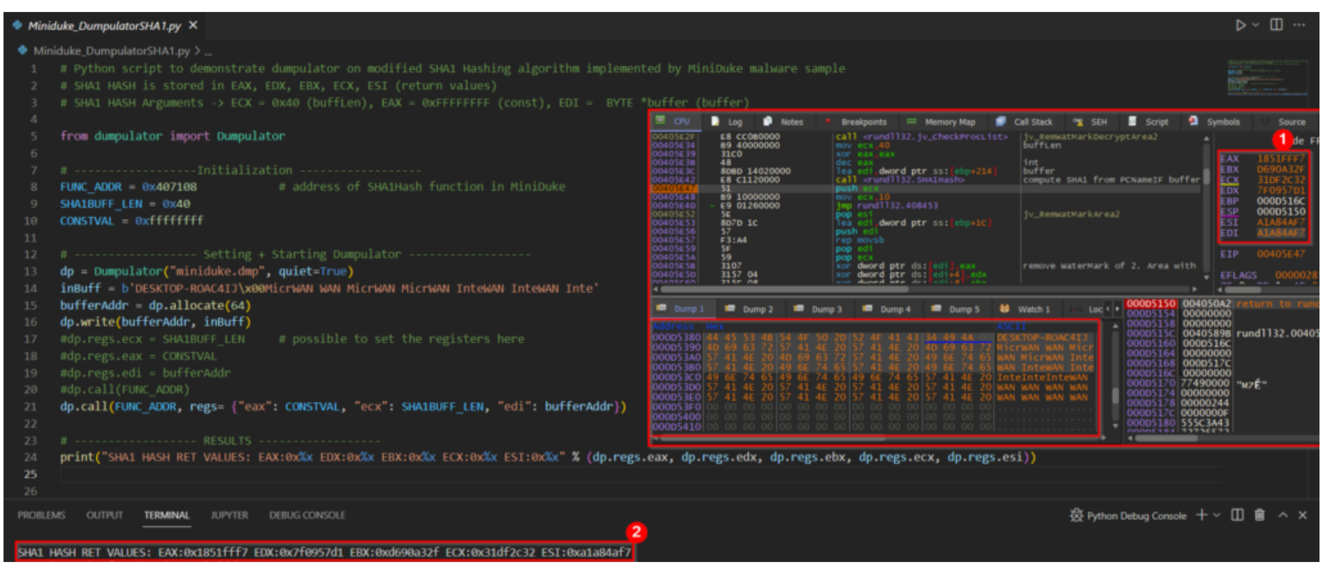


Figure 9: Script execution – “Dumpulator” producing the same SHA1 Hash values as the MiniDuke sample

Emulation – Unicorn Engine

For the emulation approach, we can use any kind of CPU emulator (ex. Qiling, Speakeasy, etc.) which is able to emulate x86 assembly and has bindings for Python language. As we do not need any higher abstraction level (Syscalls, API functions) we can use the one which most of the others are based on – Unicorn Engine.

Unicorn is a lightweight, multi-platform, multi-architecture CPU emulator framework, based on QEMU, which is implemented in pure C language with bindings for many other languages. We will be using Python bindings. Our goal is to create an independent function `SHA1Hash` which can be called like any other ordinary function in Python, producing the same SHA1 hashes as the original one in MiniDuke. The idea behind the implementation we use is pretty straightforward — we simply extract the opcode bytes of the function and use them via the CPU emulation.

Extracting all bytes of original function opcodes can be done simply via IDAPython or using IDA→Edit→Export Data.

```
# IDAPython - extracting opcode bytes of SHA1Hash function
import idaapi, idc

SHA1HashAddr = idc.get_name_ea_simple("SHA1Hash")
SHA1Hash = idaapi.get_func(SHA1HashAddr)
SHA1HASH_OPCODE = idaapi.get_bytes(SHA1Hash.start_ea, SHA1Hash.size())
SHA1HASH_OPCODE.hex()
# Output: '0f6ec589cb8dad74a3[...]'
```

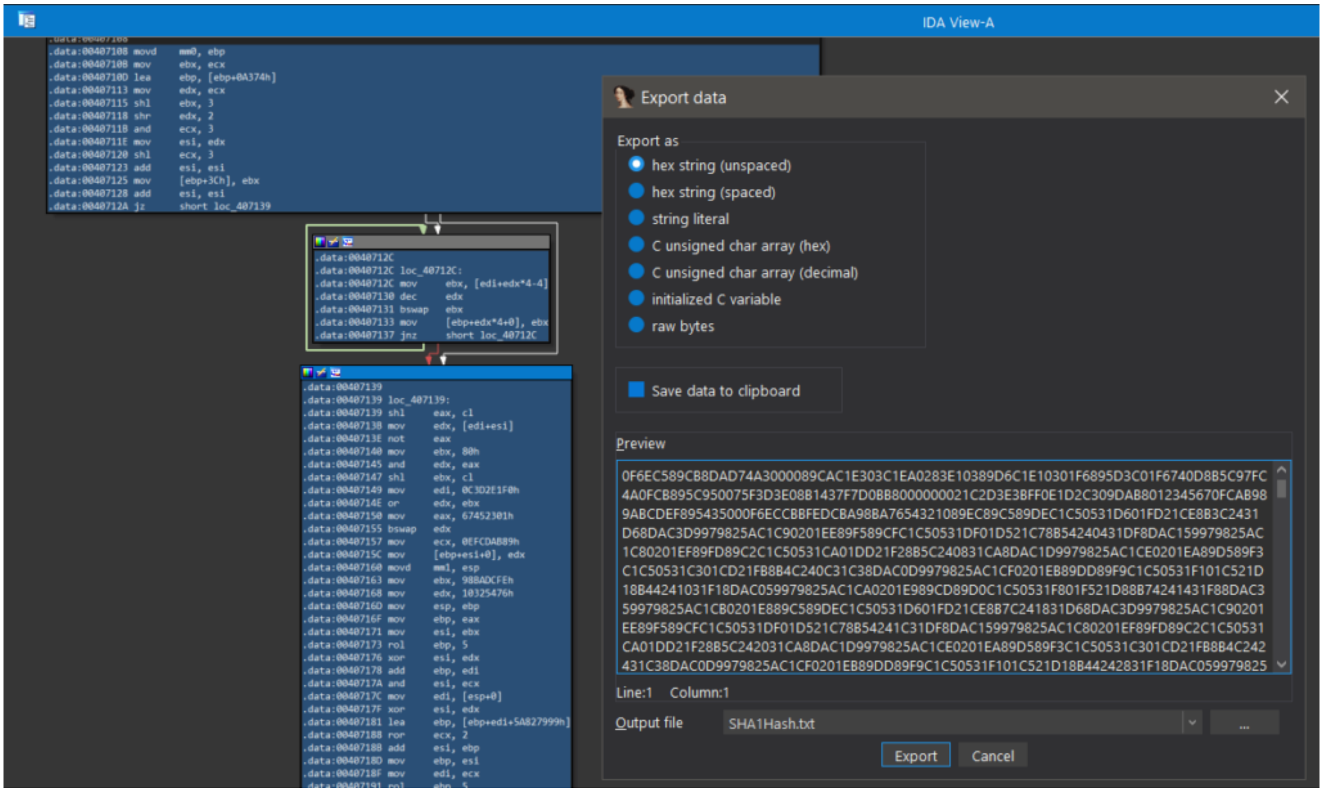


Figure 10: Using IDA "Export data" dialog to export opcode bytes of SHA1Hash function

As in the previous approaches, we need to set up the context for execution. In this case this means preparing arguments for the function, and setting addresses for our extracted opcodes and input buffer.

```
# ------------------Initialization ------------------
# remove "retn" instruction from SHA1Hash function opcodes or ->
UC_ERR_FETCH_UNMAPPED -> no ret address on stack
SHA1HASH_OPCODE = b"\x0f\x6e\xc5\x89\xcb\x8d\xad\x74\xa3.........................."
OPCODE_ADDRESS = 0x400000
SHA1BUFF_LEN = 0x40
CONSTVAL = 0xffffffff
BUFFERADDR = OPCODE_ADDRESS + 0x200000
```

Note that the last `retn` instruction should be deleted from the extracted opcode listing in order to not transfer back execution to the return address on the stack, and the stack frame should be manually set up by specifying values for `ebp` and `esp`. All these things are shown in the final Python script below.

```
# Python script to demonstrate Unicorn emulator on modified SHA1 Hashing algorithm
implemented by MiniDuke malware sample
# SHA1 HASH is stored in EAX, EDX, EBX, ECX, ESI (return values)
# SHA1 HASH Arguments -> ECX = 0x40 (buffLen), EAX = 0xFFFFFFFF (const), EDI =  BYTE
*buffer (buffer)

from unicorn import *
from unicorn.x86_const import *

def GetMinidukeSHA1(inBuff:bytes) -> Uc:
    # ------------------Initialization ------------------
    # remove "retn" instruction from SHA1Hash function opcodes or ->
UC_ERR_FETCH_UNMAPPED -> no ret address on stack
    SHA1HASH_OPCODE =
b"\x0f\x6e\xc5\x89\xcb\x8d\xad\x74\xa3..........................."
    OPCODE_ADDRESS = 0x400000
    SHA1BUFF_LEN = 0x40
    CONSTVAL = 0xffffffff
    BUFFERADDR = OPCODE_ADDRESS + 0x200000

    # ------------------ Setting + Starting Emulator ------------------
    try:
        mu = Uc(UC_ARCH_X86, UC_MODE_32)                                      # set
EMU architecture and mode
        mu.mem_map(OPCODE_ADDRESS, 0x200000, UC_PROT_ALL)                     # map
memory for SHA1Hash function opcodes, stack etc.
        mu.mem_write(OPCODE_ADDRESS, SHA1HASH_OPCODE)                         #
write opcodes to memory
        mu.mem_map(BUFFERADDR, 0x1000, UC_PROT_ALL)                           # map
memory for input to be hashed
        mu.mem_write(BUFFERADDR, inBuff)                                      #
write input bytes to memory

        mu.reg_write(UC_X86_REG_ESP, OPCODE_ADDRESS + 0x100000)               #
initialize stack (ESP)
        mu.reg_write(UC_X86_REG_EBP, OPCODE_ADDRESS + 0x100000)               #
initialize frame pointer (EBP)
        mu.reg_write(UC_X86_REG_EAX, CONSTVAL)                                # set
EAX register (argument) -> CONSTVAL
        mu.reg_write(UC_X86_REG_ECX, SHA1BUFF_LEN)                            # set
ECX register (argument) -> SHA1BUFF_LEN
        mu.reg_write(UC_X86_REG_EDI, BUFFERADDR)                              # set
EDI register (argument) -> BUFFERADDR to be hashed
        mu.emu_start(OPCODE_ADDRESS, OPCODE_ADDRESS + len(SHA1HASH_OPCODE))   #
start emulation of opcodes
        return mu

    except UcError as e:
        print("ERROR: %s" % e)

# ------------------ RESULTS ------------------
inBuff = b'DESKTOP-ROAC4IJ\x00MicrWAN WAN MicrWAN MicrWAN InteWAN InteWAN Inte'
```

```
mu = GetMinidukeSHA1(inBuff)
print("SHA1 HASH RET VALUES: EAX:0x%x EDX:0x%x EBX:0x%x ECX:0x%x ESI:0x%x" %
(mu.reg_read(UC_X86_REG_EAX), mu.reg_read(UC_X86_REG_EDX),
mu.reg_read(UC_X86_REG_EBX), mu.reg_read(UC_X86_REG_ECX),
mu.reg_read(UC_X86_REG_ESI)))
```

The script output can be seen below.

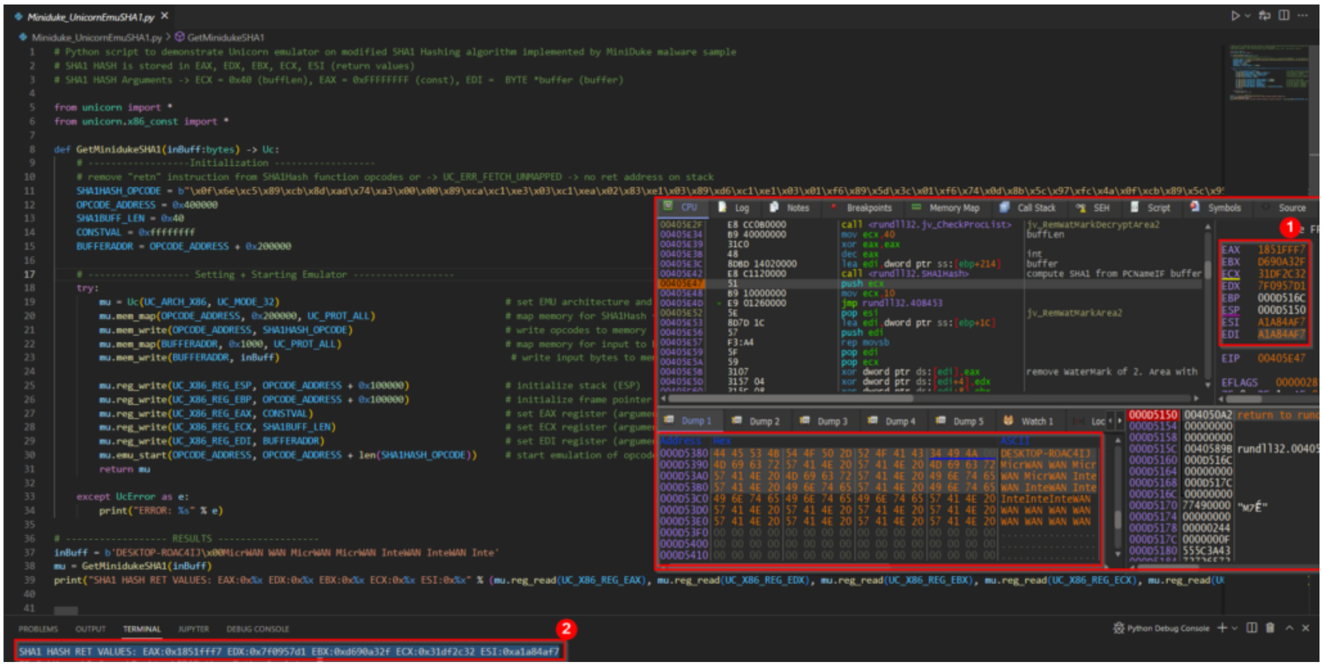


Figure 11: Script execution – "Unicorn Engine" producing the same SHA1 Hash values as the MiniDuke sample

Conclusion

All the above-described methods for direct invocation of assembly have their advantages and disadvantages. We were particularly impressed by the easy-to-use Dumpulator which is free, fast to implement, and highly effective. It is well suited for writing universal string decryptors, config extractors, and other contexts where many different logic fragments have to be called in sequence while preserving a hard-to-set-up context.

The IDA Appcall feature is one of the best solutions in situations where we would like to enrich the IDA database directly with results produced by the invocation of a specific function. Syscalls could be a part of such a function as Appcall is usually used in real execution environments – using a debugger. One of the greatest things about Appcall is the fast and easy context restoration. As Appcall relies on a debugger and could be used together with IDAPython scripting, it could even in theory be used as a basis for a fuzzer, feeding random input to functions in order to discover unexpected behavior (i.e. bugs), though the performance overhead might make this approach not very practical.

Using pure emulation via Unicorn Engine is a universal solution for the independent implementation of specific functionality. With this approach, it is possible to take a part of the code as-is and use it with no connection to the original sample. This method does not rely on a runnable sample and there is no problem to re-implement functionality for just a part of the code. This approach may be harder to implement for functions that are not a contiguous, easily-dumpable block of code. For part of code where APIs or syscalls occur, or the execution context is much harder to set up, the previously mentioned methods are usually a preferable choice.

Pros and Cons Summary

IDA Appcall

PROS:

- Natively supported by IDA
- Possible to use with `IDAPython` right in the context of IDA.
- Natively understands higher abstraction layer so Windows APIs and syscalls can be a part of the invoked function.
- Can be used on corrupted code/file with `Bochs` emulator `IDB emulate` feature (non-runnable sample).
- The combination of the `Appcall` feature and scriptable debugger is very powerful, giving us full control at any moment of Appcall execution.

CONS:

- Prototypes of more sophisticated functions using custom calling conventions (`__usercall`) are harder to implement.
- Invoked assembly needs to be a function, not just part of code.

Dumpulator

PROS:

- Very easy-to-use. Code making use of it is Pythonic and fast to implement.
- If a minidump is obtained in a state where all context is already set, with no need to map memory or set things like a stack, frame pointer, or even arguments, Dumpulator can leverage this to de-clutter the invocation code even further.
- Understands the higher abstraction layer and allows use of syscalls (though some may need to be implemented manually).
- Enables lower access level to modify the context in a similar way to usual emulation.
- Can be used to emulate part of code (does not have to be a function)

CONS:

Requires a minidump of the desired process to be worked on, which in turn requires a runnable binary sample.

Emulation – Unicorn Engine

PROS:

- The most independent solution, requires only the interesting assembly code.
- Low access level to set and modify context.
- Can be used to emulate part of code fully independently. Allows free modification and patching of instructions on the fly.

CONS:

- Harder to map memory and set the context of the emulation engine correctly.
- No out-of-the-box access to higher abstraction layer and system calls.

References

IDA – Appcall

https://hex-rays.com/blog/introducing-the-appcall-feature-in-ida-pro-5-6/

https://hex-rays.com/blog/practical-appcall-examples/

https://www.hex-rays.com/wp-content/uploads/2019/12/debugging_appcall.pdf

Dumpulator

https://github.com/mrexodia/dumpulator

https://github.com/mrexodia/MiniDumpPlugin

Unicorn Engine

https://github.com/unicorn-engine/unicorn

https://github.com/unicorn-engine/unicorn/tree/master/bindings/python

Samples + Scripts (password:infected)

1. Original MiniDuke sample: VirusTotal, miniduke_original.7z
2. Unpacked MiniDuke sample: miniduke_unpacked.7z
3. MiniDuke minidump: miniduke_minidump.7z
4. All scripts mentioned in the article: IDAPython_PythonScripts.7z