Native function and Assembly Code Invocation

research.checkpoint.com/2022/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 <u>IDA Appcall</u> 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 <u>Dumpulator</u>; and finally, we will show how to get that result using emulation with <u>Unicorn Engine</u>. The practical example used in this article is based on the "tweaked" SHA1 hashing algorithm implemented by a sample of the <u>MiniDuke</u> 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.

IDA View-A		
data:00407139		
.data:00407139 loc 407139:		: CODE XREF: SHA1Hash+22↑i
.data:00407139	shl	eax. cl
.data:0040713B	mov	edx, [edi+esi]
.data:0040713E	not	eax
.data:00407140	mov	ebx, 80h
.data:00407145	and	edx, eax
.data:00407147	shl	ebx, cl
.data:00407149	mov	edi, 0C3D2E1F0h
.data:0040714E	or	edx, ebx
.data:00407150	mov	eax, 67452301h
.data:00407155	bswap	edx
.data:00407157	mov	ecx, 0EFCDAB89h
.data:0040715C	mov	[ebp+esi+0], edx
.data:00407160	movd	mm1, esp
.data:00407163	mov	ebx, 98BADCFEh
.data:00407168	mov	edx <mark>,</mark> 10325476h
.data:0040716D	mov	esp, ebp
.data:0040716F	mov	ebp, eax
.data:00407171	mov	esi, ebx
.data:00407173	rol	ebp, 5
.data:00407176	xor	esi, edx
.data:00407178	add	ebp, edi
.data:0040717A	and	esi, ecx
.data:0040717C	mov	edi, [esp+ <mark>0</mark>]
.data:0040717F	xor	esi, edx
.data:0040/181	lea	ebp, [ebp+edi+5A827999h]

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	<pre>edi, [ebp+IFandPcNameBuff] ; buffer</pre>
call	SHA1Hash ; compute SHA1 from PCNameIF buffer

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 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)
 00000000 struc_SHA1HASH struc ; (sizeof=0x14, mappedto_138)
 00000000 EAX
                            dd ?
 00000004 EDX
                            dd ?
 00000008 _EBX_
                            dd ?
                                                                   Figure 4: IDA
 0000000C ECX
                            dd ?
 00000010 ESI
                            dd ?
 00000014 struc SHA1HASH
                            ends
 00000014
```

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 = 0xfffffff
```

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
```

.data:0040500C	; voidusercal	ll start	(int@∙	<ebx>, int@</ebx>	<esi>)</esi>				
.data:0040500C		public s	start			E.	ntry Point		
.data:0040500C	start	proc nea	ar				na yr onn		
.data:0040500C									
.data:0040500C	var_4	= dword	ptr ·	-4					
.data:0040500C									
.data:0040500C		jmp	short	t loc_40501	3				
.data:0040500E	;								
.data:0040500E									
.data:0040500E	jv_jmpDecryptAre	ea2WithC	RC32:		; CODE	E XREF: j	jv_RemWat≀	MarkArea2+15↓j	
.data:0040500E		call	jv_D∉	ecryptArea2	vithCR(C32			
.data:00405013									
.data:00405013	loc_405013:				; CODE	E XREE	startîj	adjusting	
.data:00405013		std							
.data:00405014		xor	eax,	eax					
.data:00405016		lea	edi,	[esp+var_4	I				
.data:0040501A		mov	ecx,	0AE08h					
.data:0040501F		rep sto	sd						
.data:00405021		sub	esp,	0AE08h					
.data:00405027		mov	ebp,	esp	int				
.data:00405029		cld							

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 = 0 \times 40
CONSTVAL = 0xfffffff
# -----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_)
```

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)
function entry (SHA1Hash)
idc.wait_for_next_event(idc.WFNE_SUSP, 3)
idaapi.continue_process()
to hit the new conditional breakpoint
idc.wait_for_next_event(idc.WFNE_SUSP, 3)
idc.del_bpt(bpt.ea)
conditional breakpoint
Appcall.cleanup_appcall()
conditional breakpoint -> return
```

invoking Appcall and breaking on

debugger has control now so continue

- # deleting the previously created
- # clean Appcall after hitting the

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 = 0 \times 40
CONSTVAL = 0xfffffff
# -----Set conditional BP 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
# ----- 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
                                          # clean Appcall after hitting the
Appcall.cleanup_appcall()
conditional breakpoint -> return
# ----- Exiting Debugger -----
idc.exit_process()
```

Dumpulator

Dumpulator is a python library that assists with code emulation in <u>minidump</u> files. The core emulation engine of dumpulator is based on <u>Unicorn Engine</u>, 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 (<u>x64dbg</u> – <u>MiniDumpPlugin</u>, <u>Process Explorer</u>, <u>Process Hacker</u>, Task Manager) or with the Windows API (<u>MiniDumpWriteDump</u>). We can use the <u>x64dbg</u> – <u>MiniDumpPlugin</u> to create a minidump in a state where almost all in the process is already set for SHA1 Hash creation, right before the <u>SHA1Hash</u> 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*.

• ⊃ ■ →	_II † A ⇒	🌲 🛟 📲 📓 🥒 🚍	🧳 🥒 f× #	A- 🖳 🗍 🧕		
🕮 CPU 🍃 Lo	og 📑 Notes 📍	Breakpoints 📟 Memory Ma	ip 🗐 Call Stack	😪 SEH 💆 Script	🎽 Symbols	Source
00405E2F E8 00405E34 B9 00405E39 310 00405E38 B0E 00405E32 E8 00405E32 E8 00405E32 E8 00405E42 E8 00405E42 E8 00405E42 E8 00405E43 B9 00405E54 E9 00405E55 SE 00405E56 57 00405E59 SF 00405E50 315 00405E63 314 00405E63 312 00405E63 314 00405E63 315 00405E63 316 00405E63 316 00405E63 316 00405E64 57 00405E63 316 00405E64 57 00405E66 57 00405E66 57	CC080000 4000000 0 DD 14020000 C1120000 10000000 D 1C A4 77 77 04 FF 08 FF 0C	<pre>call <rundll32.jv_checkpr mov ecx,40 xor eax,eax dec eax lea edi,dword ptr ss:[ebp call <rundll32.shaihash> push ecx mov ecx,10 jmp rundll32.408453 pop esi lea edi,dword ptr ss:[ebp push edi rep movsb pop edi pop ecx xor dword ptr ds:[edi].ea xor dword ptr ds:[edi].ea xor dword ptr ds:[edi].ea xor dword ptr ds:[edi].ea xor dword ptr ds:[edi].ea</rundll32.shaihash></rundll32.jv_checkpr </pre>	ocList> jv_Remw buffer compute +214] buffer compute jv_Remw edi:"DE edi:"DE edi:"DE edi:"DE edi:"DE edi:"DE edi:"DE edi:"DE edi:"DE edi:"DE edi:"DE	atMarkDecryptArea2 SHA1 from PCNameIF bu atMarkArea2 SKTOP-ROAC4IJ" SKTOP-ROAC4IJ" WaterMark of 2. Area w TOP-ROAC4IJ", edx:"DES ROAC4IJ" 4JJ" SKTOP-ROAC4IJ"	Affer EAX EBX ECX EDX EBP EST EDI EIP KTOP-R EFLA ZF C OF C	Hide FFFFFFF 00000000 00005380 000D516C 000D5150 00000000 000D5380 00405E42 AGS 00000296 0 PF 1 AF 1 0 SF 1 DF 0 0 TE 0 TE 1
🕮 Dump 1 📖	Dump 2 🔲 Dum	p 3 🏾 🕮 Dump 4 💭 Dump	o 5 🛛 🍪 🛛 Watch 1	📧 Locals 😕 Struct		
Address Hex 000D5380 44 45 000D5380 44 69 000D5380 57 41 000D5400 00 00	53 48 54 4F 50 63 72 57 41 4E 4E 20 4D 69 63 4E 20 4D 6E 74 74 6S 49 6E 74 4E 20 57 41 4E 4E 20 57 41 4E 4E 20 57 41 4E 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	2D 52 4F 41 43 34 49 47 20 57 41 42 04D 69 63 72 57 41 42 049 66 74 65 57 41 42 049 66 74 65 57 41 42 049 66 74 65 57 41 42 057 41 48 20 57 41 42 57 41 48 20 57 41 48 20 57 41 48 20 57 41 48 20 57 41 48 20 57 41 48 20 57 41 48 20 57 41 48 20 57 41 48 20 57 41 48 20 57 41 48 20 57 41 48 20 57 41<	ASCII DESKTOP-RO/ 72 MicrWAN WAN 4 65 WAN MicrWAN 4 65 WAN InteWAN 2 00 InteInteInt 2 20 WAN WAN WAN 2 20 WAN WAN WAN 4 000	AC4IJ N Micr N Inte Inte teWAN N WAN N WAN		
Command: MiniDur	mp "C:\Users)	\Desktop\miniduke\m	iniduke.dmp"			

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.

Ł	IDA - miniduke.dmp C:\Users	Desktop\miniduke\miniduke.	dmp				
File	le Edit Jump Search View Debug ڬ 🚍 : 🔶 🔿 : 🚰 🏪 🏪 🦀	ger Lumina Options Windows	:Help 	indba debuaaer	- ta 💼 🐗 🗎 🐔	61 * * 94	
	Library function 📕 Regular function	📕 Instruction 📕 Data 📃 Unexplo	ored 📕 External symbol 📕 Lumi	ina function			
	Debug View	×	Call Stack	× 🔄	Threads	×	Mod
	IDA View-EIP					General registers	
	rundl132:00405E2F; rundl132:00405E2F call rundl132:00405E39 mov rundl132:00405E39 mov rundl132:00405E38 dec rundl132:00405E32 call rundl132:00405E42 call rundl132:00405E47 push rundl132:00405E40 jmp rundl132:00405E40 jmp rundl132:00405E40 jmp	near ptr unk_406A00 ecx, 40h ; '@' eax, eax eax eax ecii. [ebp+214h] sub_407108 ecx, 10h loc_408453 SE42 (Synchronized with EIP)		Î	*EAX FFFFFFF *EBX 00000040 /* b'@`*/ *ECX 0000040 /* b'@`*/ *EDX 00005380 (debug008) *ESI 00000080 *ESP 00005380 (debug008) *ESP 00005180 -> 00405002 *ESP 00005150 -> 00405002 *ESP 00005542 (rund1132) *EFL 00000296	-> ("DESKTOP-ROAC4I]") -> ("DESKTOP-ROAC4I]") -> 00000000 -> 0000118 (debug008) -> call near ptr unk_407;	-> 00000000 108
\bigcirc	Hex View-1		- ø	× 💿	Stack view		× 🚺
000 000 000 000 000 000 000	2005380 44 45 53 48 54 4F 50 2D 2005380 4D 69 63 72 57 41 4E 20 2005380 57 41 4E 20 4D 69 63 72 2005380 57 41 4E 20 4D 69 63 72 2005380 57 41 4E 20 49 6E 74 65 2005300 57 41 4E 20 57 41 4E 20 2005310 57 41 4E 20 57 41 4E 20 2005310 57 41 4E 20 57 41 4E 20 2005310 57 41 4E 20 57 41 4E 20 2005310 57 41 4E 20 57 41 4E 20	52 4F 41 43 44 94 00 57 41 4E 20 40 69 63 72 57 41 4E 20 49 6E 74 65 57 41 4E 20 49 6E 74 65 57 41 4E 20 49 6E 74 65 49 6E 74 65 57 41 4E 20 57 41 4E 20 57 41 4E 20 08 00 00 00	DESKTOP-ROAC4IJ. MicrWAN-WAN-Micr WAN-MicrWAN-Inte WAN-InteWAN-Inte InteInteInteWAN- WAN-WAN-WAN-WAN-WAN- WAN-WAN-WAN-WAN-WAN-	 00:0000 000051 01:0004 000051 02:0008 000051 03:000C 000051 04:0010 000051 05:0014 000051 06:0018 000051 07:001C 000051 	50 -> 004050A2 -> 000011E8 54 (debug008) -> 00000000 58 (debug008) -> 00000000 50 (debug008) -> 00000000 60 -> 0040516C (debug008) 64 (debug008) -> 00000000 68 -> 0000517C (debug008) 5C (debug008) -> 00000000	(debug008) -> 00000000 -> 04246C88 -> 00000000 -> 0000000F	

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 = 0 \times 407108
                        # address of SHA1Hash function in MiniDuke
SHA1BUFF_LEN = 0 \times 40
CONSTVAL = 0xfffffff
# ------ 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.

🔹 Minia	luke_DumpulatorSHA1.py ×						⊳ ~ ⊞ …
🔹 Min	Iduke_DumpulatorSHA1.py >						
1	# Python script to demonstrate dumpulator on modified SHA1 Hashing algorithm implement		Duke malware s				
2	# SHA1 HASH is stored in EAX, EDX, EBX, ECX, ESI (return values)						
3	# SHA1 HASH Arguments -> ECX = 0x40 (bufflen), EAX = 0xFFFFFFFF (const), EDI = BYTE						
4		🗏 CPU 🛛	Log 👂 Notes	* Breakpoints = Memory Map	🖉 Call Stack 🕿 SEH 📓 Scrip	t 💁 Symbo	is Source
5	from dumpulator import Dumpulator	00405£2F	E8 CC080000	call <rund1132.jv_checkproclist< td=""><td>jv_RemwatMarkDecryptArea2</td><td></td><td></td></rund1132.jv_checkproclist<>	jv_RemwatMarkDecryptArea2		
6		00405E34 00405E39	89 40000000 31C0		BuffLen		1851007
7			48 8080 14020000	dec eax lea_edi_dword_ptr_ss:[ebp+214]	int buffer	EB	D690A32F
8	FUNC_ADDR = 0x407108 # address of SHA1Hash function in MiniDuke		E8 C1120000 51	call <rund1132.sha1hash> push ecx</rund1132.sha1hash>	compute SHA1 from PCNameIF	buffer EC	310F2C32 7F0957D1
9		00405E48 00405E4D	89 10000000 E9 01260000			EB	P 000D516C
10		00405E52 00405E53	5E 807D 1C	pop esi lea edi,dword ptr ss:[ebp+1C]			I A1A84AF7
11			\$7 F3:A4			ED	1 A1A84AF7
12			5F 59	pop edi pop ecx		ET	00405E47
13	<pre>dp = Dumpulator("miniduke.dmp", quiet=True)</pre>		3107 3157 04	xor dword ptr ds: edl, eax xor dword ptr ds: edl+4],edx		with EF	LAGS 0000028
14	<pre>inBuff = b'DESKTOP-ROAC4IJ\x00MicrWAN WAN MicrWAN MicrWAN InteWAN Inte'</pre>	a a a a a a a a a a a a a a a a a a a	3150 08	use dured into destruit of the			
15	bufferAddr = dp.allocate(64)	💷 Dump 1	💷 Dump 2 🐖	Dump 3 📁 Dump 4 🐖 Dump 5	Watch 1 Loc 4 . 000051	0 004050A2	return to rund
16	dp.write(bufferAddr, inBuff)				ASCI1 000051	8 00000000	
17		00005380		50 2D 52 4F 41 43 34 49 4A 00 F	ESKTOP-ROAC4IJ 000051	C 00405898 0 0000516C	rund1132.00405
18		000D53A0		63 72 57 41 4E 20 49 6E 74 65 F	AN MicrwAN Inte 000051	4 00000000	
19		00005380	41 4E 20 49 6E 9 6E 74 65 49 6E	74 65 57 41 4E 20 49 6E 74 65 9 74 65 49 6E 74 65 57 41 4E 20	AN InteWAN Inte 000051	00000000	
20		00005300 5		4E 20 57 41 4E 20 57 41 4E 20 Y	IAN WAN WAN WAN 000051	4 00000000	"M7E"
21	<pre>dp.call(FUNC_ADOR, regs= {"eax": CONSTVAL, "ecx": SHA1BUFF_LEN, "edi": bufferAddr})</pre>	000D53F0 0			000051	8 00000244 7C 0000000F	
22		00005400				0 555C3A43	
23							
24	print("SHA1 HASH RET VALUES: EAX:0x%x EDX:0x%x EBX:0x%x ECX:0x%x ESI:0x%x" % (dp.regs	eax, dp.re	egs.edx, dp.reg	s.ebx, dp.regs.ecx, dp.regs.	esi))		
25							
26							
PROBLEM	MS OUTPUT TERMINAL JUPYTER DEBUG CONSOLE				Python Debug Co	nsole + ~ 1	
	2	4					
							/

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 – <u>Unicorn Engine</u>.

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[...]'
```

19		IDA View-A
.dra:00407108 movd mm0, ebp .dra:00407108 movd ebp, ecx .dra:00407108 mov ebp, ecx .dra:00407110 nov edx, ecx .dra:00407113 shl ebx, 3 .dra:00407118 shd ecx, 3 .dra:00407118 and ecx, 3 .dra:00407118 and ecx, 3		Export data
.data:00407120 shl ecx, 3 .data:00407123 db esi, esi .data:00407125 mov [ebp=3Ch], ebx .data:00407128 db esi, esi .data:00407128 jz short loc_407139	bx 7/339 	 hex string (unspaced) hex string (spaced) string literal C unsigned char array (hex) C unsigned char array (decimal) initialized C variable raw bytes
	.dst::08407140 mov ebx, 80h .dst::08407145 and edx, eax .dst::08407147 shl ebx, cl .dst::08407146 mov edt, e0x .dst::08407156 mov edx, ebx .dst::08407156 mov edx, ebx .dst::08407155 mov .dst::08407156 mov edx, ebx .dst::08407168 mov edx, 18225476h .dst::08407168 mov edx, 18225476h .dst::08407168 mov edx, 18225476h .dst::08407168 mov edx, 18225476h .dst::08407176 mov edx, 1825476h .dst::08407176 mov edx, 1825476h .dst::08407178 mov edt, 18, edx .dst::08407178 mov edt, 18, edx .dst::08407188	OF6EC589CB8DAD74A3000089CAC1E303C1EA0283E10389D6C1E10301F6895D3C01F6740D885C97FC 4A0FC8895C950075F3D3E0881437F7D0B88000000021C2D3E38FF0E1D2C309DA88012345670FCAB98 9A8CDEF895435000F6ECC8BFEDC8A988A7654321089EC89C589DEC1C50531D601FD21CE883C2431 D68DAC3D9979825AC1C90201EE89F589CFC1C50531D601D521C78854240431D88DAC39979825AC1 C1C50531C301CD21F889F820C1C50531C401D521E27885C240831CA8DAC109979825AC1C60201E889D89F9 18844241031F18DAC059979825AC1CA0201E989CD89D0C1C50531F01521D8874244431F88DAC3 59978825AC1C60201E88959990EC1C50531D601FD21C8887C241831F88DAC3 59978825AC1C60201E8895099DEC1C50531D601FD21C8887C241831F88DAC3 59979825AC1C60201E8895099DEC1C50531F001D521C7885424031D601FD21C8887C241831F88DAC3 59979825AC1C60201E8895099DEC1C50531F1001C521D 18844241031F18DAC059979825AC1C60201E8892D899DC1C50531F001D521C78854241C31DF88DAC309979825AC1C60201 E8895895C420031CA8DAC1D9979825AC1C60201EA89D589F3C1C50531C301C021F889FB825AC1C60201 E8995895C41C009979825AC1C60201EA89D589F3C1C50531C301C021F889FD89C202 Val1C38DAC0D9979825AC1C60201EA89D589F3C1C50531C301C021F889F25 Line:1 Column:1 Qutput file SHA1Hash.txt

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 = 0xfffffff
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 = 0 \times 400000
    SHA1BUFF_LEN = 0 \times 40
    CONSTVAL = 0xfffffff
    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.req_write(UC_X86_REG_ESP, OPCODE_ADDRESS + 0x100000)
                                                                             #
initialize stack (ESP)
       mu.req_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.





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
 (<u>__usercall</u>) 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: <u>IDAPython_PythonScripts.7z</u>