The Last Stage of Delirium
Research Group

Microsoft Windows RPC Security Vulnerabilities

HITB Security Conference
December 12th, 2003

Presentation overview

- Introduction to Microsoft RPC
- Reverse engineering of Microsoft RPC services
  - dmidl (reverse midl)
  - fa (reverse c)
- Exploitation techniques for RPC vulnerabilities
  - RPC DCOM RemoteActivation (stack overflow)
  - RPC Messenger (heap overflow)
- Summary

Part 1:

Introduction to Microsoft RPC

It's 106 miles to Chicago, we've got a full tank of gas, half a pack of cigarettes, it's dark and we're wearing sunglasses.

-- Elwood Blues

Introduction to Microsoft RPC

What is it?

Remote Procedure Call (RPC) is an inter-process communication mechanism that allows client and server software to communicate over the network

There are two main standards of RPC mechanism:

- DCE (Distributed Computing Environment) RPC
- ONC (Open Network Computing) RPC

Microsoft RPC is compatible with the Open Group's Distributed Computing Environment specification for remote procedure calls

Introduction to Microsoft RPC
Communication mechanisms

The Last Stage of Delirium Research Group

Microsoft RPC uses IPC mechanisms, such as named pipes, LPC ports, NetBIOS, or Winsock, to establish communications between the client and the server

RPC servers can be reached with the use of different RPC, transport and network protocols (*protocol-sequence*)

A given RPC server may listen for requests on multiple endpoints, which are specific to the registered protocol-sequence

Introduction to Microsoft RPC
Communication mechanisms (2)

Protocol sequences supported by Microsoft RPC:

ncacn_nb_tcp	Connection-oriented NetBIOS over Transmission Control Protocol (TCP)
ncacn_nb_ipx	Connection-oriented NetBIOS over Internet Packet Exchange (IPX)
ncacn_nb_nb	Connection-oriented NetBIOS Enhanced User Interface (NetBEUI)
ncacn_ip_tcp	Connection-oriented Transmission Control Protocol/Internet Protocol (TCP/IP)
ncacn_np	Connection-oriented named pipes
ncacn_spx	Connection-oriented Sequenced Packet Exchange (SPX)
ncacn_dnet_nsp	Connection-oriented DECnet transport
ncacn_at_dsp	Connection-oriented AppleTalk DSP
ncacn_vns_spp	Connection-oriented Vines scalable parallel processing (SPP) transport
ncadg_ip_udp	Connectionless User Datagram Protocol/Internet Protocol (UDP/IP)
ncadg_ipx	Connectionless IPX
ncadg_mq	Connectionless over the Microsoft® Message Queue Server (MSMQ)
ncacn_http	Connection-oriented TCP/IP using Internet Information Server as HTTP proxy
ncalrpc	Local procedure call

RPC client/server architecture

Specific functionality of a given RPC server is exposed in a form of interfaces identified by their identifiers (UUID) and version (major and minor) numbers

Each interface can contain a set of functions that can be called remotely

Before a call to a given RPC function, an appropriate BIND operation must be issued in order to uniquely assign client application to the target RPC interface with which it wants to talk to

Introduction to Microsoft RPC

Why it is so important ?

Microsoft RPC has been a backbone communication mechanism used in Windows operating system since its early days (Windows NT 3.1, back in 1993)

There are many (if not all) Windows services that heavily rely on the RPC infrastructure:

- services expose their functionality through MS RPC
- RPC interfaces of a service can be very often reached remotely (either through ncacn_ip_tcp, ncadg_ip_udp or ncacn_np), what means that successful bind operation can be issued on them

RPC interfaces (Windows 2000)

The Last Stage of Delirium Research Group

RPC interfaces that can be by default reached remotely on Windows 2000 systems (SP4 + all hotfixes) through ncacn_np:

```
12345678-1234-abcd-ef00-0123456789ab v1.0 (spoolsv.exe)
12345778-1234-abcd-ef00-0123456789ab v0.0 (lsasrv.dll)
c681d488-d850-11d0-8c52-00c04fd90f7e v1.0 (lsasrv.dll)
3919286a-b10c-11d0-9ba8-00c04fd92ef5 v0.0 (lsasrv.dll)
12345778-1234-abcd-ef00-0123456789ac v1.0 (samsrv.dll)
d335b8f6-cb31-11d0-b0f9-006097ba4e54 v1.5 (polagent.dll)
98fe2c90-a542-11d0-a4ef-00a0c9062910 v1.0 (advapi32.dll)
367abb81-9844-35f1-ad32-98f038001003 v2.0 (services.exe)
93149ca2-973b-11d1-8c39-00c04fb984f9 v0.0 (scesrv.dll)
82273fdc-e32a-18c3-3f78-827929dc23ea v0.0 (eventlog.dll)
65a93890-fab9-43a3-b2a5-1e330ac28f11 v2.0 (dnsrslvr.dll)
8d9f4e40-a03d-11ce-8f69-08003e30051b v1.0 (umpnpmgr.dll)
4b324fc8-1670-01d3-1278-5a47bf6ee188 v3.0 (srvsvc.dll)
6bffd098-a112-3610-9833-46c3f87e345a v1.0 (wkssvc.dll)
8d0ffe72-d252-11d0-bf8f-00c04fd9126b v1.0 (cryptsvc.dll)
c9378ff1-16f7-11d0-a0b2-00aa0061426a v1.0 (cryptsvc.dll)
0d72a7d4-6148-11d1-b4aa-00c04fb66ea0 v1.0 (cryptsvc.dll)
6bffd098-a112-3610-9833-012892020162 v0.0 (browser.dll)
17fdd703-1827-4e34-79d4-24a55c53bb37 v1.0 (msgsvc.dll)
300f3532-38cc-11d0-a3f0-0020af6b0add v1.2 (trkwks.dll)
3ba0ffc0-93fc-11d0-a4ec-00a0c9062910 v1.0 (wmicore.dll)
```

Introduction to Microsoft RPC
RPC interfaces (Windows 2000) cont.

RPC interfaces that can be by default reached remotely on Windows 2000 systems (SP4 + all hotfixes) through ncacn_ip_tcp:

```
e1af8308-5d1f-11c9-91a4-08002b14a0fa v3.0 (rpcss.dll)
0b0a6584-9e0f-11cf-a3cf-00805f68cb1b v1.1 (rpcss.dll)
975201b0-59ca-11d0-a8d5-00a0c90d8051 v1.0 (rpcss.dll)
e60c73e6-88f9-11cf-9af1-0020af6e72f4 v2.0 (rpcss.dll)
99fcfec4-5260-101b-bbcb-00aa0021347a v0.0 (rpcss.dll)
b9e79e60-3d52-11ce-aaa1-00006901293f v0.2 (rpcss.dll)
412f241e-c12a-11ce-abff-0020af6e7a17 v0.2 (rpcss.dll)
00000136-0000-0000-c000-000000000046 v0.0 (rpcss.dll)
c6f3ee72-ce7e-11d1-b71e-00c04fc3111a v1.0 (rpcss.dll)
4d9f4ab8-7d1c-11cf-861e-0020af6e7c57 v0.0 (rpcss.dll)
000001a0-0000-0000-c000-000000000046 v0.0 (rpcss.dll)
1ff70682-0a51-30e8-076d-740be8cee98b v1.0 (mstask.exe)
378e52b0-c0a9-11cf-822d-00aa0051e40f v1.0 (mstask.exe)
```

Introduction to Microsoft RPC

RPC interfaces (Windows XP)

RPC interfaces that can be by default reached remotely on Windows XP systems (SP1 + all hotfixes) through ncacn_np:

```
12345778-1234-abcd-ef00-0123456789ab v0.0 (lsasrv.dll)
621dff68-3c39-4c6c-aae3-e68e2c6503ad v1.0 (wzcsvc.dll)
18f70770-8e64-11cf-9af1-0020af6e72f4 v0.0 (ole32.dll)
1ff70682-0a51-30e8-076d-740be8cee98b v1.0 (schedsvc.dll)
378e52b0-c0a9-11cf-822d-00aa0051e40f v1.0 (schedsvc.dll)
0a74ef1c-41a4-4e06-83ae-dc74fb1cdd53 v1.0 (schedsvc.dll)
3faf4738-3a21-4307-b46c-fdda9bb8c0d5 v1.0 (audiosrv.dll)
6bffd098-a112-3610-9833-46c3f87e345a v1.0 (wkssvc.dll)
8d0ffe72-d252-11d0-bf8f-00c04fd9126b v1.0 (cryptsvc.dll)
a3b749b1-e3d0-4967-a521-124055d1c37d v1.0 (cryptsvc.dll)
0d72a7d4-6148-11d1-b4aa-00c04fb66ea0 v1.0 (cryptsvc.dll)
f50aac00-c7f3-428e-a022-a6b71bfb9d43 v1.0 (cryptsvc.dll)
12b81e99-f207-4a4c-85d3-77b42f76fd14 v1.0 (seclogon.dll)
8fb6d884-2388-11d0-8c35-00c04fda2795 v4.1 (w32time.dll)
300f3532-38cc-11d0-a3f0-0020af6b0add v1.2 (trkwks.dll)
63fbe424-2029-11d1-8db8-00aa004abd5e v1.0 (sens.dll)
629b9f66-556c-11d1-8dd2-00aa004abd5e v3.0 (sens.dll)
4b324fc8-1670-01d3-1278-5a47bf6ee188 v3.0 (srvsvc.dll)
3f77b086-3a17-11d3-9166-00c04f688e28 v1.0 (srvsvc.dll)
17fdd703-1827-4e34-79d4-24a55c53bb37 v1.0 (msgsvc.dll)
6bffd098-a112-3610-9833-012892020162 v0.0 (browser.dll)
5ca4a760-ebb1-11cf-8611-00a0245420ed v1.0 (termsrv.dll)
000001a0-0000-0000-c000-000000000046 v0.0 (rpcss.dll)
```

Introduction to Microsoft RPC

RPC interfaces (Windows XP) cont.

The Last Stage of Delirium Research Group

RPC interfaces that can be by default reached remotely on Windows XP systems (SP1 + all hotfixes) through ncacn_ip_tcp:

```
e1af8308-5d1f-11c9-91a4-08002b14a0fa v3.0 (rpcss.dll)
0b0a6584-9e0f-11cf-a3cf-00805f68cb1b v1.1 (rpcss.dll)
1d55b526-c137-46c5-ab79-638f2a68e869 v1.0 (rpcss.dll)
e60c73e6-88f9-11cf-9af1-0020af6e72f4 v2.0 (rpcss.dll)
99fcfec4-5260-101b-bbcb-00aa0021347a v0.0 (rpcss.dll)
b9e79e60-3d52-11ce-aaa1-00006901293f v0.2 (rpcss.dll)
412f241e-c12a-11ce-abff-0020af6e7a17 v0.2 (rpcss.dll)
00000136-0000-0000-c000-000000000046 v0.0 (rpcss.dll)
c6f3ee72-ce7e-11d1-b71e-00c04fc3111a v1.0 (rpcss.dll)
4d9f4ab8-7d1c-11cf-861e-0020af6e7c57 v0.0 (rpcss.dll)
000001a0-0000-0000-c000-000000000046 v0.0 (rpcss.dll)
621dff68-3c39-4c6c-aae3-e68e2c6503ad v1.0 (wzcsvc.dll)
18f70770-8e64-11cf-9af1-0020af6e72f4 v0.0 (ole32.dll)
1ff70682-0a51-30e8-076d-740be8cee98b v1.0 (schedsvc.dll)
378e52b0-c0a9-11cf-822d-00aa0051e40f v1.0 (schedsvc.dll)
0a74ef1c-41a4-4e06-83ae-dc74fb1cdd53 v1.0 (schedsvc.dll)
3faf4738-3a21-4307-b46c-fdda9bb8c0d5 v1.0 (audiosrv.dll)
6bffd098-a112-3610-9833-46c3f87e345a v1.0 (wkssvc.dll)
12b81e99-f207-4a4c-85d3-77b42f76fd14 v1.0 (seclogon.dll)
```

RPC interfaces (XP) cont.

The Last Stage of Delirium Research Group

RPC interfaces that can be by default reached remotely on Windows XP systems (SP1 + all hotfixes) through ncacn_ip_tcp:

```
8fb6d884-2388-11d0-8c35-00c04fda2795 v4.1 (w32time.dll)
300f3532-38cc-11d0-a3f0-0020af6b0add v1.2 (trkwks.dll)
8d0ffe72-d252-11d0-bf8f-00c04fd9126b v1.0 (cryptsvc.dll)
a3b749b1-e3d0-4967-a521-124055d1c37d v1.0 (cryptsvc.dll)
0d72a7d4-6148-11d1-b4aa-00c04fb66ea0 v1.0 (cryptsvc.dll)
f50aac00-c7f3-428e-a022-a6b71bfb9d43 v1.0 (cryptsvc.dll)
63fbe424-2029-11d1-8db8-00aa004abd5e v1.0 (sens.dll)
629b9f66-556c-11d1-8dd2-00aa004abd5e v3.0 (sens.dll)
4b324fc8-1670-01d3-1278-5a47bf6ee188 v3.0 (srvsvc.dll)
3f77b086-3a17-11d3-9166-00c04f688e28 v1.0 (srvsvc.dll)
17fdd703-1827-4e34-79d4-24a55c53bb37 v1.0 (msgsvc.dll)
6bffd098-a112-3610-9833-012892020162 v0.0 (browser.dll)
5ca4a760-ebb1-11cf-8611-00a0245420ed v1.0 (termsrv.dll)
000001a0-0000-0000-c000-000000000046 v0.0 (rpcss.dll)
```

Other RPC interfaces

There are many more RPC interfaces in Windows 2000/XP system. These interfaces can be divided respectively into:

- interfaces that can be only reached locally either through ncacn_np or ncalrpc protocol sequences
- ORPC interfaces, which require proper OBJREF pointer for the call to proceed (usually obtained through IRemoteActivation interface)
- interfaces introduced to the system along with a specific application (i.e. Microsoft Internet Information Services, Microsoft Exchange, Microsoft SQL Server, ...)

More details: Windows Network Services Internals, J.B. Marchand
http://www.hsc.fr/ressources/articles/win_net_srv/index.html.en

Introduction to Microsoft RPC

Authentication issues

Presented Windows interfaces can be reached from the network through ncacn_np protocol sequence and *NULL SESSION*

Reachability (successful BIND operation) does not necessarily mean that functions of a given interface can be actually called (!) as there are some server applications that restrict access to its interfaces on a per-client basis by defining a security-callback function (RpcServerRegisterIfEx).

RpcServerRegisterAuthInfo function can be used for defining what authentication service to use when the server receives a request for a remote procedure call

RPC server may use the RpcBindingInqAuthClient function to check whether the client connection meets the desired level of authentication.

Authorization issues

Most interfaces run with SYSTEM privileges and impersonate the client for the time of processing its request (RpcImpersonateClient)

If the server code has an implementation flaw that may lead to the code execution, SYSTEM privileges can be always reestablished by issuing a call to RpcRevertToSelf (regardless of the privileges possessed at the time of the call)

In some cases, client privileges are additionally checked after impersonation (i.e. OpenThreadToken/PrivilegeCheck/CheckTokenMembership call sequence)

RPC runtime security issues

If there are multiple RPC interfaces registered in one process:

- Each of them can be reached through any of the protocol sequences registered in that process,
- Context handles from one interface are valid and can be passed to the other completely unrelated interface (unless strict_context_handle attribute is used for the interface)

If the server stub was compiled without the /robust switch, RPC marshaler may not reject all malformed RPC packets

Additionally, if the [range] keyword is not used in an IDL interface definition file, RPC interface may accept requests to access out-of-bounds data

Reference: Writing Secure Code, Second Edition, M. Howard, D. LeBlanc
http://www.amazon.com

Introduction to Microsoft RPC
Example service

```
void *my_malloc(int size){
    return(HeapAlloc(GetProcessHeap(),0,size));
}

int func_1(handle_t h,int i,struct s *stab[],unsigned char *str){
    char* p;
    hyper a;

    if(!(p=my_malloc(32))){
        return(1);
    }
    lstrcpy(p,str);
    return(0);
}
```

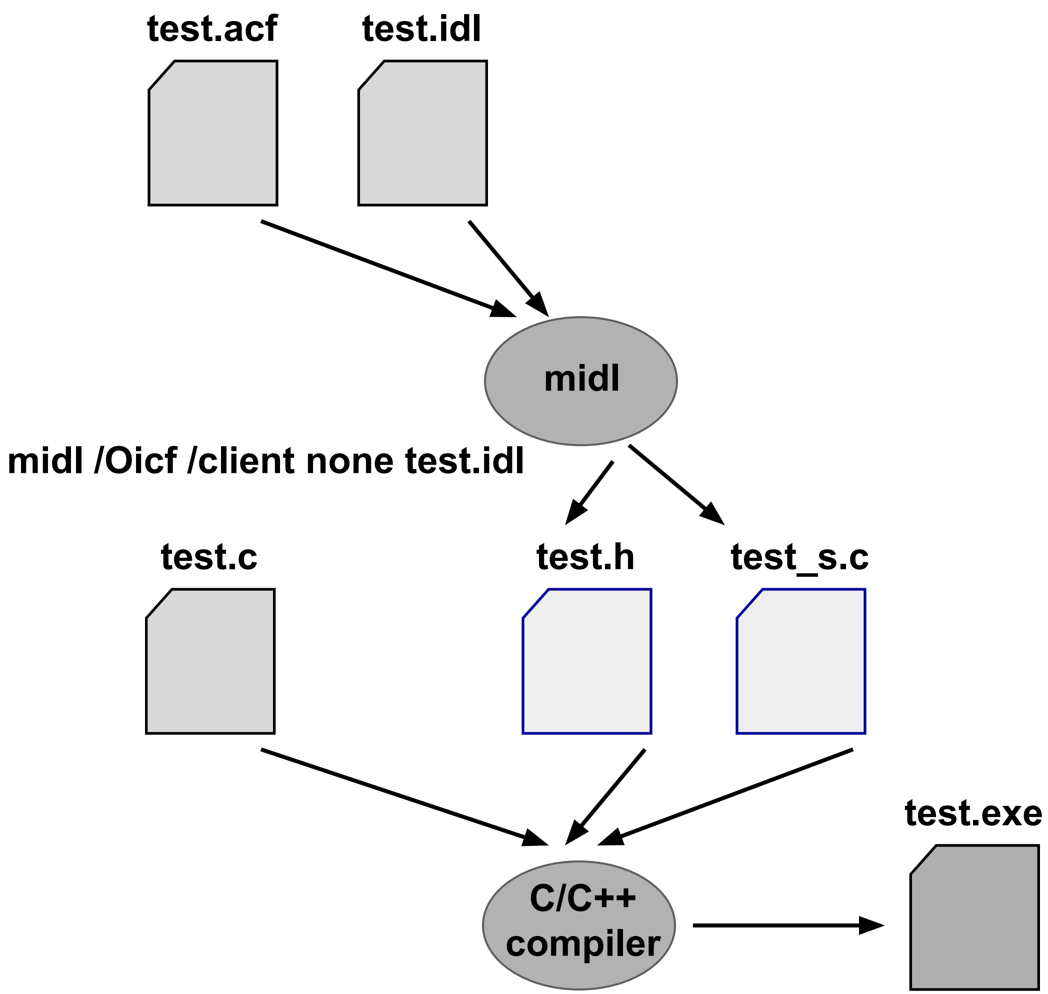
Introduction to Microsoft RPC
Interface Definition (IDL)

The Last Stage of Delirium Research Group

```
[
  uuid(11111111-2222-3333-4444-555555555555),
  version(1.0)
]

interface if{

  struct s{
    byte b;
    hyper h;
  };

  int func_1(
    [in] handle_t h,
    [in] int i,
    [out,size_is(i)] struct s *stab[],
    [in,string,size_is(256)] char *c
  );

}
```

Introduction to Microsoft RPC

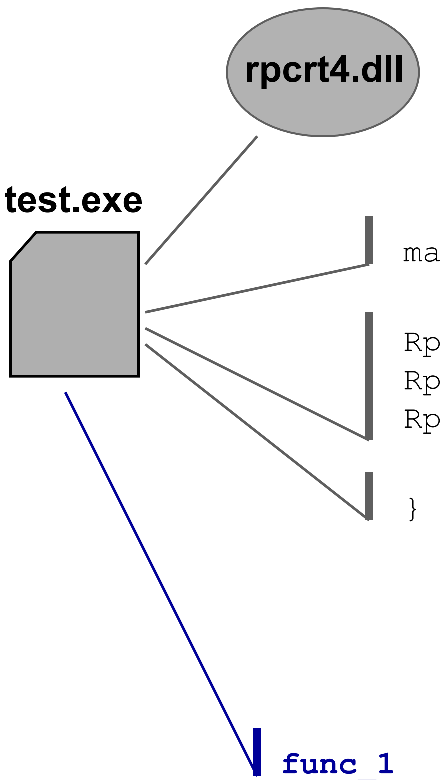
Midl compiler (midl.exe)



Introduction to Microsoft RPC

RPC/NDR engine (rpcrt4.dll)

The Last Stage of Delirium Research Group




```
main(int argc,char **argv){

RpcServerUseProtseqEp(prot,5,endp,NULL);
RpcServerRegisterIf(if_v1_0_s_ifspec,NULL,NULL);
RpcServerListen(1,1234,FALSE);

}
```

The Last Stage of Delirium
Research Group

Part 2:

Reverse engineering of Microsoft RPC

Basic research is when I'm doing what I don't know what I'm doing.

-- Wernher Von Braun

dmidl (reverse MIDL)

RPC interface decompiler

The Last Stage of Delirium Research Group

Dmidl is a tool that reverse RPC interfaces definitions build with the use of Microsoft IDL compiler. It performs automatic search for binaries that contains MIDL generated stubs and tries to decompile them back to IDL

Dmidl supports fully-interpreted (/Oi and /Oicf) as well as mixed (/Os) marshaling modes. It was tested on Windows 2000, XP and 2003 binaries

The tool was written in 2001 by reverse engineering midl.exe binary and comparing/analysing files generated by this compiler. Later, in 2002, it was updated according to more detailed NDR documentation published in MSDN

Another midl decompiler: muddle, M. Chapman

http://www.cse.unsw.edu.au/~matthewc/muddle/

dmidl (reverse MIDL)
How it works

- Finding and parsing RPC control structures
- Reversing procedure format strings
- Reversing type format strings
- Combining parameter and type information
- Generating interface definition (.idl file)

```
z:\projects\DMIDL-2.0>dmidl -g idl.test2
rpc interface decompiler (reverse midl) [version 2.0]
copyright LAST STAGE OF DELIRIUM 2001-2002 poland  //lsd-pl.net/

idl.test2

11111111-2222-3333-4444-555555555555 v1.0  test-oi.exe.1.idl        1 stub
11111111-2222-3333-4444-555555555555 v1.0  test-oicf.exe.1.idl      1 stub
11111111-2222-3333-4444-555555555555 v1.0  test-os.exe.1.idl        1 stub

12 files analysed, 3 interfaces found

z:\projects\DMIDL-2.0>
```

Finding and parsing RPC control structures
/Oicf and /Oi modes

The Last Stage of Delirium
Research Group

```
struct RPC_SERVER_INTERFACE{
    RPC_SYNTAX_IDENTIFIER InterfaceId;      = 11111111-2222-3333-4444-555555555555, v 1.0
    RPC_SYNTAX_IDENTIFIER TransferId;       = 045d888a-eb1c-c911-9fe8-08002b104860, v 2.0
    RPC_DISPATCH_TABLE *DispatchTable;      -> RPC_DISPATCH_FUNCTION table[]
    ...                                        NdrServerCall2 (/Oicf)
    MIDL_SERVER_INFO *ServerInfo               NdrServerCall (/Oi)
};
```

```
struct MIDL_SERVER_INFO{
    MIDL_STUB_DESC *StubDesc;
    SERVER_ROUTINE *DispatchTable;
    FORMAT_STRING *ProcFormatString;
    short *FormatStringOffset;
    ...
};
```

```
struct MIDL_STUB_DESC{
    char *TypeFormatString;
    long Version;             = 0x20000 (/Oicf)
    ...                       = 0x10001 (/Oi)
};
```

```
SERVER_ROUTINE table[]
func1
func2
```

Finding and parsing RPC control structures

/Os mode

```
struct RPC_SERVER_INTERFACE{
    RPC_SYNTAX_IDENTIFIER InterfaceId;      = 11111111-2222-3333-4444-555555555555, v 1.0
    RPC_SYNTAX_IDENTIFIER TransferId;       = 045d888a-eb1c-c911-9fe8-08002b104860, v 2.0
    RPC_DISPATCH_TABLE *DispatchTable;      ->  RPC_DISPATCH_FUNCTION table[]
    ...                                             if_func1
    MIDL_SERVER_INFO *ServerInfo            = NULL  if_func2
};
```

```
void __RPC_STUB if_func2(RPC_MESSAGE *RpcMessage){
    NdrServerInitializeNew(
        RpcMessage,&StubMsg,&StubDesc
    );
    NdrConvert(
        &StubMsg,&ProcFormatString.Format[24]
    );
    func1(...);
}
```

FormatStringOffset

```
struct MIDL_STUB_DESC{
    char *TypeFormatString;
    long Version;                           = 0x10001
    ...
};
```

Reversing procedure format strings
/Oicf mode

```
FUNCTIONS:

func_1

00000: 00                                   handle_type
00001: 48                                   old_flags
00002: 00 00 00 00                          rpc_flags
00006: 00 00                                method_index 0
00008: 14 00                                stack_size 20
00010: 32 00 00 00                          explicit_handle
00014: 08 00                                in_param_hint 8
00016: 08 00                                out_param_hint 8
00018: 07                                   oi2_flags
00019: 04                                   cparams 4
00020: 48 00 04 00 08 00                    in FC_LONG
00026: 13 00 08 00 0a 00                    in -> 00010
00032: 0b 01 0c 00 2c 00                    out -> 00044
00038: 70 00 10 00 08 00                    in ref FC_LONG
```

Reversing procedure format strings
/Oi and /Os modes

```
FUNCTIONS:

func_1

00000: 00                               handle_type
00001: 48                               old_flags
00002: 00 00 00 00                      rpc_flags
00006: 00 00                            method_index 0
00008: 14 00                            stack_size 20
00010: 32 00 00 00                      explicit_handle
00014: 4e 0f                            in FC_IGNORE
00016: 4e 08                            in FC_LONG
00018: 51 01 0a 00                      out -> 00010
00022: 4d 01 28 00                      in -> 00040
00026: 53 08                            return FC_LONG

FUNCTIONS:

func_1

00000: 4e 0f                            in FC_IGNORE
00002: 4e 08                            in FC_LONG
00004: 51 01 0a 00                      out -> 00010
00008: 4d 01 28 00                      in -> 00040
00012: 53 08                            return FC_LONG
```

Reversing type format strings
Initial decoding

```
TYPES:

00002: 15                              FC_STRUCT
00003: 07                                align 8
00004: 10 00                             size 16
00006: 01                                FC_BYTE
00007: 39                                FC_ALIGNM8
00008: 0b                                FC_HYPER
00009: 5b                              FC_END

00010: 1b                              FC_CARRAY
00011: 03                                align 4
00012: 04 00                             size 4
00014: 28 00 00 00                       size_is
00018: 4b 5c                             FC_PP
00020: 48 49 04 00 00 00 01 00             FC_VARIABLE_REPEAT
00028: 00 00 00 00 12 00 e0 ff               FC_UP -> 00002
00036: 5b                                FC_END
00037: 08                                FC_LONG
00038: 5c                                FC_PAD
00039: 5b                              FC_END

00040: 11 00 02 00                     FC_RP -> 00044

00044: 22 44 40 00 00 01               FC_C_CSTRING
```

Recognized types:

- base types
- strings
- structures
- unions
- arrays
- pointers
- other

Combining parameter and type information

Complex types

- Enumerate implicit/explicit handles and contexts
- Follow embedded types and pointers
- Calculate stack positions, offsets, alignments and padding values for fields in structures and unions
- Analyze correlation descriptors and fields' attributes
- Enumerate known callback functions (x86 opcode pattern matching)

The Last Stage of Delirium Research Group

Generating interface definition

.IDL file

```
[
  uuid(11111111-2222-3333-4444-555555555555),
  version(1.0)
]

interface if{

  /* TYPES */

  struct _2{
    byte _1;
    hyper _2;
  };

  /* FUNCTIONS */

  long
  func_1(
    /* adr 0x00401000 sym ? */

    [in] handle_t _1,
    [in] long _2,
    [out,size_is(_2)] struct _2 *_3[],
    [in,ref,size_is(256),string] char *_4
  );
}
```

An interface definition generated by dmidl is compatible with midl compiler and may be recompiled

Identified RPC function names are resolved with the use of Windows symbol files (dbghelp.dll library)

FA – Win32 x86 code decompiler

Why to decompile code?

Manual analysis of even medium size machine level code functions is usually very difficult, tiring and it takes lots of time. This is mainly due to the fact that machine level code usually:

- Introduces lots of redundant instructions (i.e. PUSH/POP)
- Is optimized with regard to memory accesses, conditional instructions, subroutine invocations
- Lacks lots of information with regard to subroutines, function arguments, return values and local variables
- Lacks type information
- Lacks information about the original code structure (loops, if/else blocks)

FA – Win32 x86 code decompiler

Why to decompile code? (2)

The Last Stage of Delirium Research Group

The process of code decompilation allows to obtain some high level code (syntax similar to C) that is much more informative for the security auditor than the original machine code

The FA project was started in January 2003 for the purpose of decompiling RPC interfaces from the Windows operating system binary files. Currently it allows for:

- Dumping RPC interface information from the target binary
- Disassembling selected function from a given RPC interface
- Decompiling selected function from a given RPC interface into C-like language

FA – Win32 x86 code decompiler

Dumping RPC interface information

```
z:\projects\FA>fa -p test.exe
rpc interface decompiler (reverse c) [version 0.9]
copyright LAST STAGE OF DELIRIUM 2003 poland  //lsd-pl.net/
image:   test.exe
.code:   0x66001000-0x66004000 (12288 bytes)
.data:   0x66004000-0x66006000 (8192 bytes)
.idata:  0x66004000-0x660040b0
RPC interfaces:
    [ 0] 11111111-2222-3333-4444555555555555 ver. 1.0
      func_0 0x66001018
```

FA – Win32 x86 code decompiler
Disassembling RPC function

The Last Stage of Delirium Research Group

```
z:\projects\FA>fa test.exe -d 0 0
rpc interface decompiler (reverse c) [version 0.9]
copyright LAST STAGE OF DELIRIUM 2003 poland  //lsd-pl.net/
image:  test.exe
disassembling from 0x66001018

66001000        PUSH    ebp
66001001        MOV     ebp,esp
66001003        MOV     eax,dword ptr [ebp+8]
66001006        PUSH    eax
66001007        PUSH    0
66001009        CALL    GetProcessHeap
6600100f        PUSH    eax
66001010        CALL    HeapAlloc
66001016        POP     ebp
66001017        RET
entry:
66001018        PUSH    ebp
66001019        MOV     ebp,esp
6600101b        SUB     esp,c
6600101e        PUSH    20
66001020        CALL    loc_66001000
66001025        ADD     esp,4

66001028        MOV     dword ptr [ebp+fffffffc],eax
6600102b        CMP     dword ptr [ebp+fffffffc],0
6600102f        JNE     loc_66001038
66001031        MOV     eax,1
66001036        JMP     loc_66001048
66001038        MOV     eax,dword ptr [ebp+14]
6600103b        PUSH    eax
6600103c        MOV     ecx,dword ptr [ebp+fffffffc]
6600103f        PUSH    ecx
66001040        CALL    lstrcpyA
66001046        XOR     eax,eax
66001048        MOV     esp,ebp
6600104a        POP     ebp
6600104b        RET
```

FA – Win32 x86 code decompiler

Decompiling RPC function

The Last Stage of Delirium Research Group

```
z:\projects\FA>fa test.exe -w 0 0
rpc interface decompiler (reverse c) [version 0.9]
copyright LAST STAGE OF DELIRIUM 2003 poland  //lsd-pl.net/
image:  test.exe
loading type info from windows.h
decompiling from 0x66001018
...
LPVOID __cdecl sub_66001000(SIZE_T arg1) {
               return HeapAlloc(GetProcessHeap(),0,arg1)
}

int __cdecl entry_66001018(unknown arg1,unknown arg2,unknown arg3,LPCSTR arg1) {
/* frame: type=ebp, size=12
   local vars:
   LPCSTR loc2 (ebp offset -4, size 4)
*/
               loc2 = sub_66001000(20)
               if (loc2<>0) {
                eax = lstrcpyA(loc2,arg1)
                eax = 0
               } else {
                eax = 1
               }
               return eax
}
```

FA – Win32 x86 code decompiler

Decompiler operation

In general, the process of FA operation is a reverse of the compilation process (but to be true it is much simpler)

FA works in several passes:

- Code disassembly, subroutines and call tree enumeration
- Compiler idioms and inline calls detection
- Conversion to high level language, push/pop removal
- Subroutine arguments and local vars enumeration
- Operands merging, dead operands removal
- Code structuring – finding loops and if/else constructs in code
- Type propagation

FA – Win32 x86 code decompiler
Decompiler features

The Last Stage of Delirium Research Group

Current version of FA is able to:

- Convert machine level code into a set of 10 high level codes (ASSIGN, TRY/EXCEPT, CALL, GOTO, RET, IF, SWITCH, QMARK, WHILE, FOR)
- Structure code (find loops and if/else constructs, regardless of their nesting)
- Locate inline calls and compiler idioms in the machine code (C operator ?, inline memset, memcpy, strlen, strchr, etc.)
- Find out information about function arguments, local variables and in most cases about their types
- Work against optimized code (shared instructions, very tricky)
- Remove redundant information from code (removing unused instructions, merging operands expressions)

FA – Win32 x86 code decompiler

Decompiler features (2)

On average FA is able to reduce the size of code to analyze after decompilation about 60% (counted in the number of instructions)

It usually allows to find out what a given function actually does

FA can use PDB/DBG info (if available) to produce much more readable code

It proved very well as it was used for locating MS03-026 and MS03-043 vulnerabilities and some other flaws that had been fixed in the meantime ;-)

Part 3:

Exploitation techniques for RPC vulnerabilities

If I had only known, I would have been a locksmith.

-- Albert Einstein

RPC vulnerabilities
Exploitation details

Phases:

- Invoking remote RPC function (TCP and UDP)
- Jumping to specified memory location
- Finding user data in process memory
- Executing user supplied code
- Avoiding process crash (and Windows reboot)

Special:

- Bypassing Windows 2003 stack overflow detection

RPC DCOM RemoteActivation service

MS03-026

- The vulnerability exists in the RemoteActivation function exported by the 4d9f4ab8-7d1c-11cf-861e0020af6e7c57 RPC interface
- Server implementing this interface is located in rpcss.dll image. It is loaded into the address space of the svchost process which is started by default on any Win2000/XP/2003 system
- Successful exploitation of the vulnerability results in a remote code execution with the highest (SYSTEM) privileges in the target Windows operating system

Invoking remote RPC function (TCP)

RemoteActivation()

The Last Stage of Delirium Research Group

IDL specification

```
error_status_t
RemoteActivation(
  [in] handle_t _1,
  [in,ref] struct _110 *_2,
  [out,ref] struct _144 *_3,
  [in,ref] struct _20 *_4,
  [in,unique,string] wchar_t *_5,
  [in,unique] struct _188 *_6,
  [in] long _7,
  [in] long _8,
  [in] long _9,
  [in,unique,size_is(_9)] struct _20 *_10,
  [in] short _11,
  [in,size_is(_11)] short _12[],
  [out,ref] hyper *_13,
  [out,ref] struct _252 **_14,
  [out,ref] struct _20 *_15,
  [out,ref] long *_16,
  [out,ref] struct _6 *_17,
  [out,ref] long *_18,
  [out,ref,size_is(_9)] struct _188 **_19,
  [out,ref,size_is(_9)] long *_20
);
```

The vulnerability results from a buffer overrun condition in a GetMachineName() function, which copies user provided wchar_t* argument passed to the RemoteActivation() function to the fixed-length local stack buffer

Invoking remote RPC function (TCP)
BIND packet

The Last Stage of Delirium Research Group

ofs	hex code	fields
00:	05 00	rpc version (5)
02:	0b	packet type (BIND)
03:	03	flags
04:	10 00 00 00	encoding
08:	?? ??	frag len
0a:	00 00	auth len
0c:	00 00 00 00	call id
10:	00 00	max xmit frag
12:	00 00	max recv frag
14:	00 00 00 00	
18:	01 00 00 00	
1c:	01 00 00 00	
20:	b8 4a 9f 4d 1c 7d cf 11	IFID = 4d9f4ab8-7d1c-11cf-861e-0020af6e7c57
28:	86 1e 00 20 af 6e 7c 57	
30:	00 00 00 00	vers = v0.0
34:	04 5d 88 8a eb 1c c9 11	TSID
3c:	9f e8 08 00 2b 10 48 60	
44:	02 00 00 00	vers

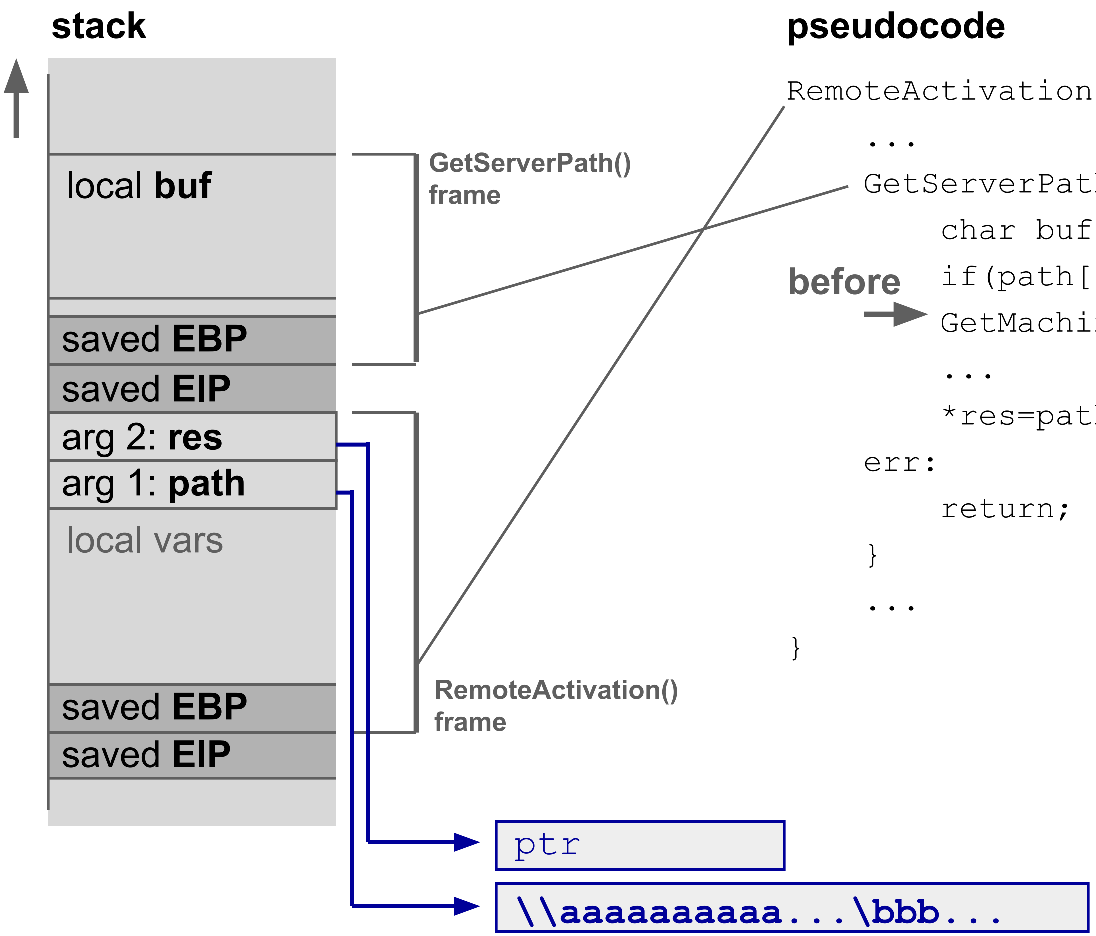
Invoking remote RPC function (TCP)
REQUEST packet

The Last Stage of Delirium Research Group

ofs	hex code	fields
00:	05 00	rpc version (5)
02:	00	packet type (REQUEST)
03:	03	flags
04:	10 00 00 00	encoding
08:	?? ??	frag len
0a:	00 00	auth len
0c:	00 00 00 00	call id
10:	00 00	max xmit frag
12:	00 00	max recv frag
14:	00 00 00 00	
18:	05 00 02 00 01 00	arg 2: struct _110 * = {{5,2},1,0,0,0}
	...	
48:	01 00 00 00	arg 5: wchar_t * = "\\aaaaa\bb"
4c:	01 00 00 00	
50:	01 00 00 00	
54:	61 61 61 61 ...	string
	...	
??:	01 00 00 00	arg 7:
??:	01 00 00 00	arg 8:
??:	01 00 00 00	arg 9:

Jumping to specified memory location

Original stack frames



pseudocode

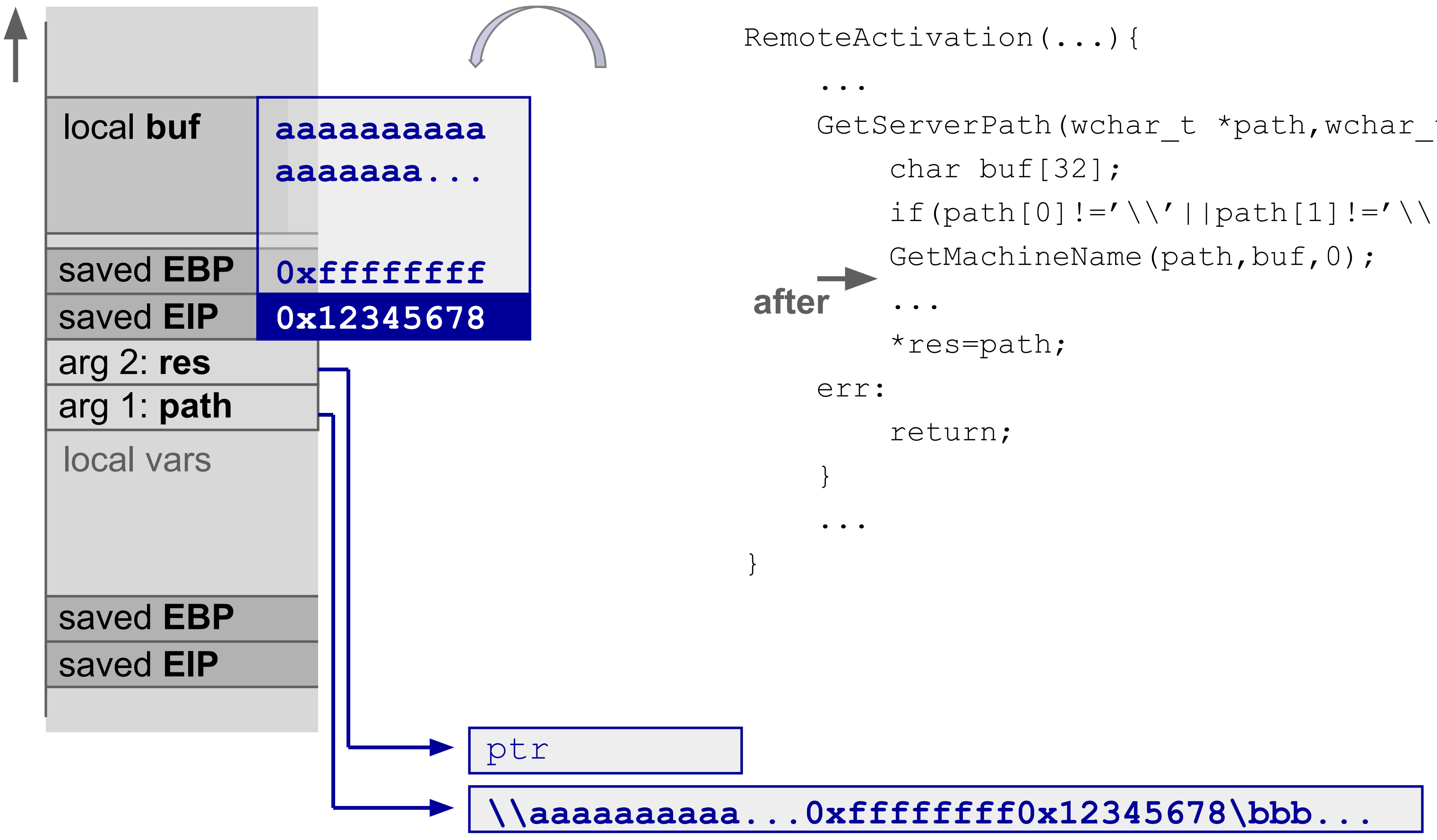
```
RemoteActivation(...){
    ...
    GetServerPath(wchar_t *path,wchar_t **res){
        char buf[32];
        if(path[0]!='\\'||path[1]!='\\') goto err;
        GetMachineName(path,buf,0);
        ...
        *res=path;
    err:
        return;
    }
    ...
}
```

before →

Jumping to specified memory location

Stack frames after buffer overflow

stack



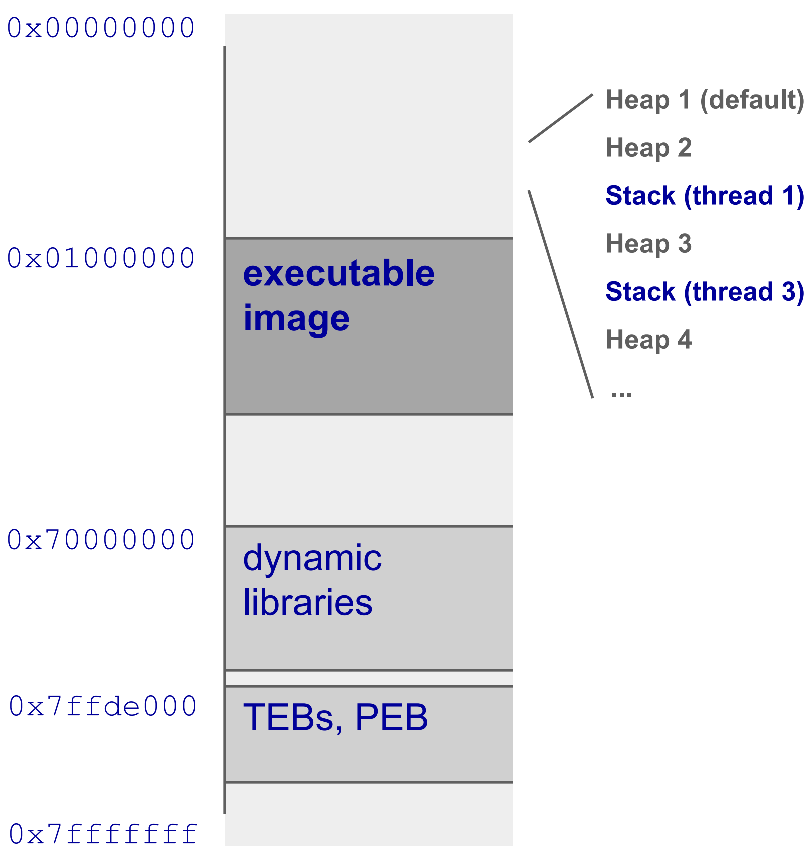
pseudocode

```
RemoteActivation(...){
    ...
    GetServerPath(wchar_t *path,wchar_t **res){
        char buf[32];
        if(path[0]!='\\'||path[1]!='\\') goto err;
        GetMachineName(path,buf,0);
        ...
        *res=path;
    err:
        return;
    }
    ...
}
```

Finding user data in process memory

Process address space

svchost process memory map



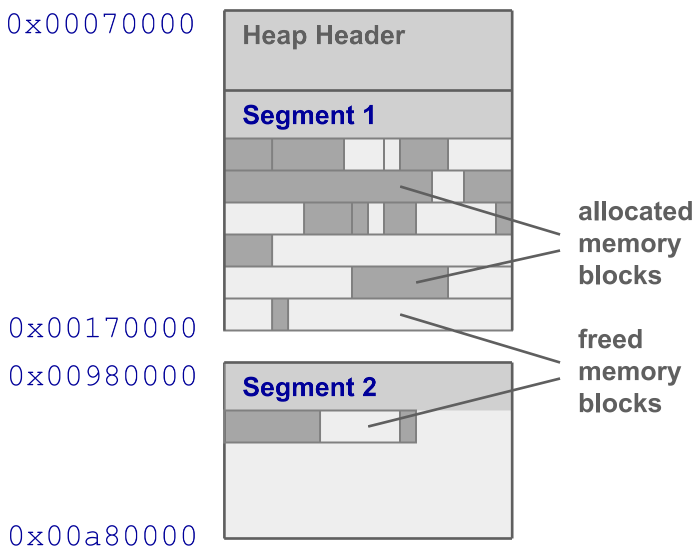
- The most difficult problem that occurs during remote exploitation of the bug on Windows 2000/XP/2003 is finding the address of memory location, where dynamically allocated, user provided data (containing asmcode) resides

- This is primarily caused by the fact that heap and stack areas, base addresses, executable and libraries images are different across different operating systems versions, service packs and languages

- This also results from the fact that vulnerable components are multithreaded

Finding user data in process memory
Heap layout

svchost default process heap



NOTE:

addresses of allocated memory blocks are hard to predict especially in the case of multithreaded processes

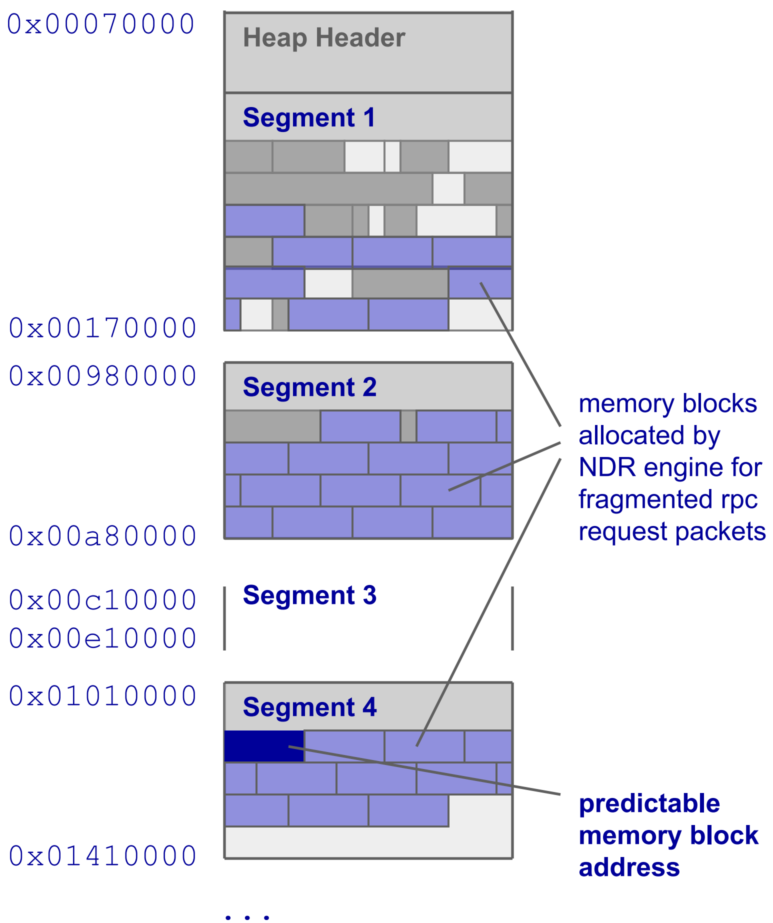
- Every process has one default heap (in svchost it starts at 0x70000), which has one linear memory segment
- If more memory space is required by an application, the Heap manager can request additional segments from the operating system
- Position and size of segments depends on virtual process memory maps (thus the application, libraries it uses etc)
- Freed memory blocks are concatenated (whenever possible) and are available for further allocation
- With time, available memory space is fragmented

Finding user data in process memory

Filling the Heap in linear way

The Last Stage of Delirium Research Group

svchost default process heap



- The goal is to fill up the remote process address space in a linear way
- RPC packet fragmentation mechanism may be used to send data that will be allocated on Heap
- When there are no more free blocks, Heap manager enlarges the existing segment by requesting new memory pages directly from OS. If this is not sufficient, it allocates memory space for new segments
- New segments are allocated in highly predictable addresses
- About 10-15 MB of data send to remote machine will place given data at the address that is constant for every version of Windows 2000 and XP (0x01080080)

Finding user data in process memory

OTHER METHODS

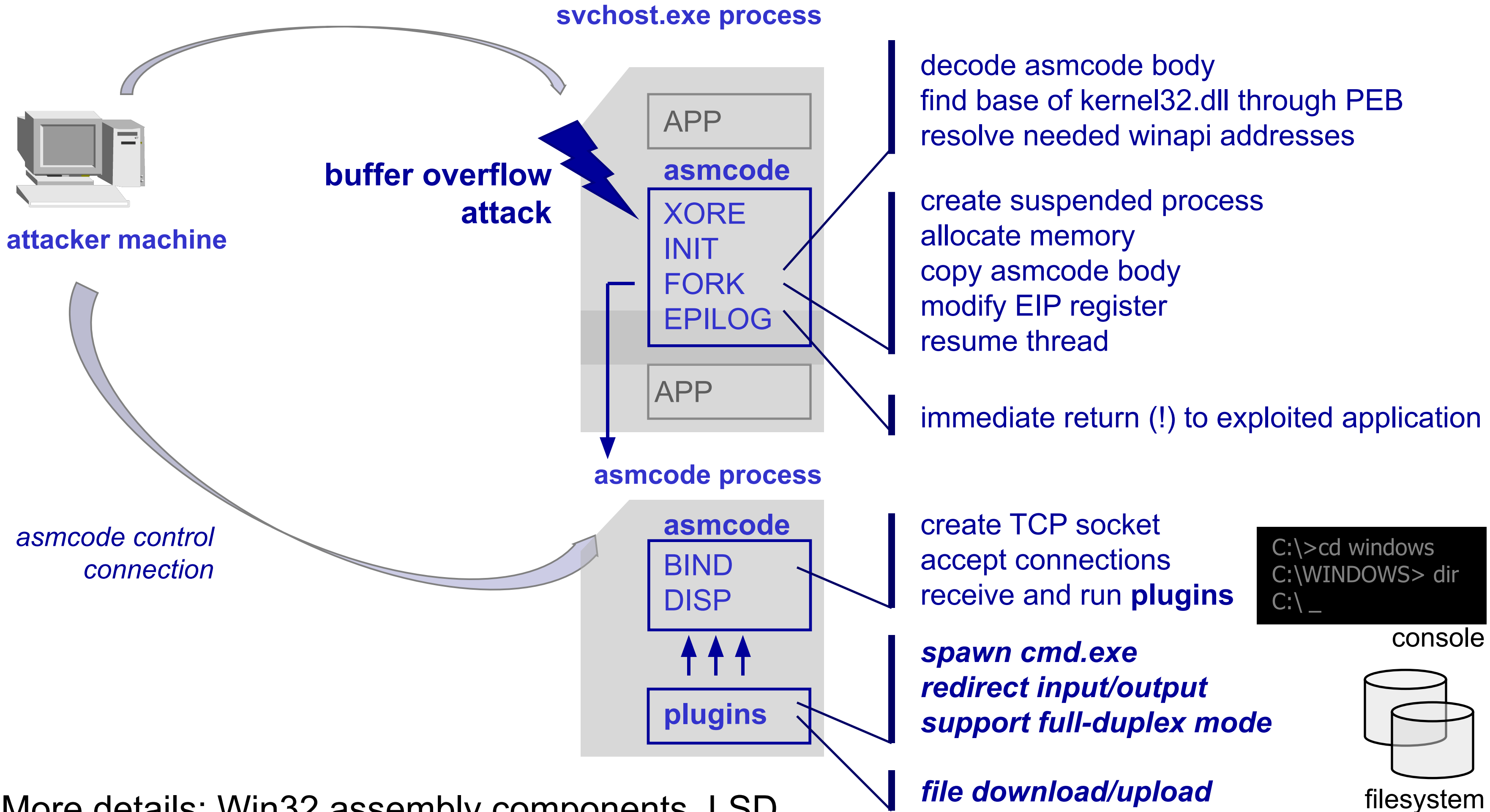
- Relative jump through call ebx instruction stored in code segment of svchost.exe executable image may be used
- After return from GetServerPath() function ebx register points to the overwritten stack frame
- svchost.exe image base address and call instruction offset do not depend on installed service pack or operating system language version
- 3 universal addresses for Windows 2000, XP, 2003
- Windows versions may be easily distinguished if communication with rpc services is possible

Reference: dcom proof of concept code, .:[oc192.us]:. Security
http://packetstormsecurity.nl/0308-exploits/oc192-dcom.c

The Last Stage of Delirium Research Group

Executing user supplied code

WINASM



More details: Win32 assembly components, LSD

http://www.lsd-pl.net/windows_components.html

Avoiding process crash
Roll back on SEH

The Last Stage of Delirium Research Group

- svchost process is very critical for Windows operating system and cannot be terminated or stopped, as it might easily lead to the system malfunction and unavoidable reboot
- Structure Exception Handling mechanism may be used to restore stable state of svchost process after stack overflow attack
- In order to do it, a special instruction sequence is executed to generate an divide by zero exception
- Exception is caught by the operating system and gets handled by the exception frame common for every function executed remotely through RPC engine
- Handler performs stack unwind operation, restores registers' contents and resumes process execution

Avoiding process crash

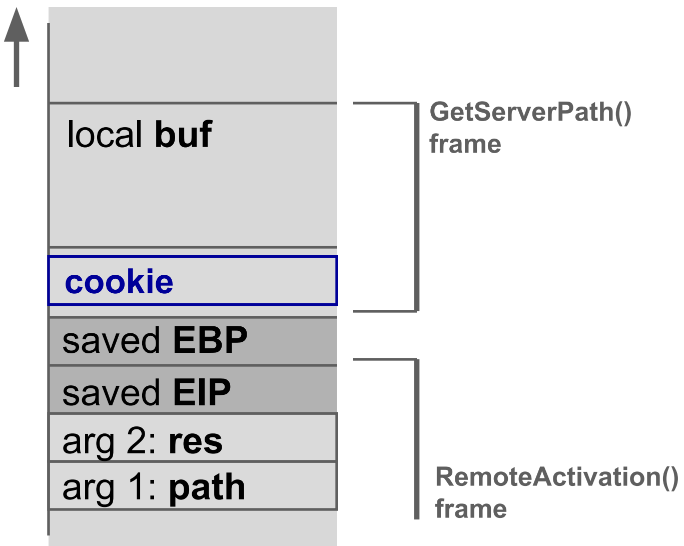
OTHER METHODS

- An alternative way to stabilize svchost process after an attack is to use ExitThread() function
- By using call to this function, a process crash can be avoided because the thread that has corrupted stack in result of buffer overflow is terminated
- Using this method, an attack on the same process may be performed multiple times, as NDR engine creates new thread for the purpose of new RPC requests
- This approach slightly changes the behavior of svchost process however it does not corrupt its operating

Reference: dcom proof of concept code, .:[oc192.us]:. Security
http://packetstormsecurity.nl/0308-exploits/oc192-dcom.c

Bypassing Windows 2003 stack bo detection
The idea of Visual C /GS switch

stack



prolog

```
push  ebp
mov   ebp,esp
sub   esp,28h
mov   eax,[__security_cookie]
mov   [ebp+0ch],eax
```

epilog

```
mov   ecx,[ebp+0ch]
call  __security_check_cookie
leave
retn  8
```

If the cookie was unchanged, ***__security_check_cookie*** executes the RET instruction and ends the function call. If the cookie doesn't match, it calls report_failure, which calls error_handler.

```
void __security_error_handler(int code,void *data){
    if(user_handler!=NULL) user_handler(code, data);
    else {__crtMessageBoxA();_exit(3);}
}
```

Reference: Compiler Security Checks In Depth, B. Bray (MSFT)

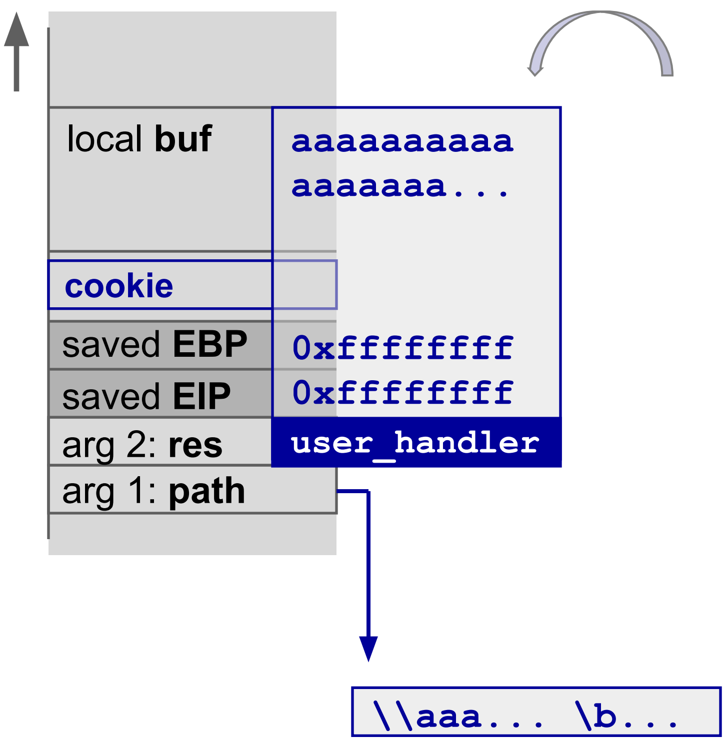
http://www.codeproject.com/tips/seccheck.asp

The Last Stage of Delirium Research Group

Bypassing Windows 2003 stack guard protection

Overwriting user_handler

stack



pseudocode

```
RemoteActivation(...){
    ...
    GetServerPath(wchar_t *path,wchar_t **res){
        char buf[32];
        if(path[0]!='\\'||path[1]!='\\') goto err;
        GetMachineName(path,buf,0);
after →
        ...
        *res=path;
    err:
        return;
    }
    ...
}
```

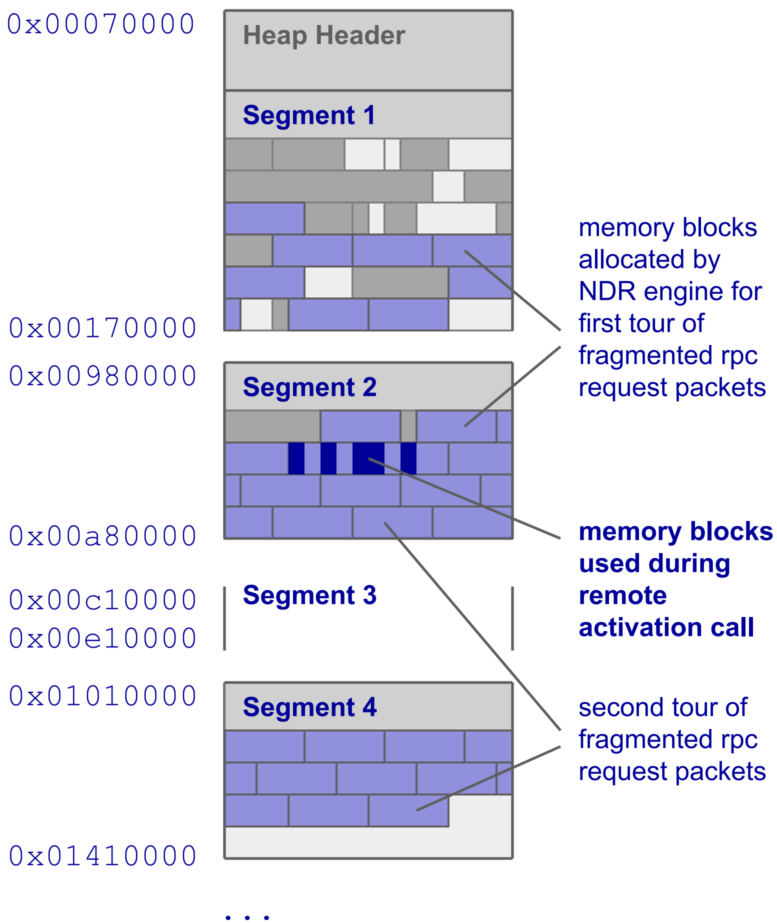
```
mov     eax,[user_handler]
mov     [eax],path
...
mov     ecx,[ebp+0ch]
cmp     ecx,[__security_cookie]
jnz     raport_failure
...
call    [user_handler]
```

Reference: Microsoft Compiler Flaw Technical Note, C. Ren, M. Weber, and G. McGraw
http://www.cigital.com/news/index.php?pg=art&artid=70

Bypassing Windows 2003 stack guard protection

Jump to \\aaa...\b... obstacle

svchost default process heap



hex code	x86 instruction opcodes
5c	pop esp
00 5c 00 61	add **[eax+eax+61]**,bl
...	

- Establish 15 parallel TCP connections
- For each of them send 6000 packets (1024 bytes long) and call remote activation method (no overflow)
- Send next 160000 packets to properly fill up remaining memory space
- Invoke remote activation method in the way that would trigger buffer overflow

RPC bcache will reuse blocks allocated during first call and eax register will point to them

Bypassing Windows 2003 stack guard protection

OTHER METHODS

- Structure Exception Handling mechanism may be used
- The idea is to modify exception registration structure located on the stack when performing buffer overflow
- Next step is to trigger an exception before security cookie check is made (by writing beyond the stack)
- Overwritten pointer to exception handler must point to an address outside the address space of loaded module (jump through register instruction)

Reference: Defeating the Stack Based Buffer Overflow Prevention Mechanism of Microsoft Windows 2003 Server, D. Litchfield

http://www.nextgenss.com/papers/defeating-w2k3-stack-protection.pdf

RPC messenger service

MS03-43

- The vulnerability exists in the NetrSendMessage function exported by the 5a7b91f8-ff00-11d0-a9b2-00c04fb6e6fc RPC interface
- Server implementing this interface is located in msgsvc.dll image. It is loaded into the address space of the svchost process, which is started by default on any Windows 2000/XP system. On Windows 2003 messenger service is disabled by default
- Successful exploitation of the vulnerability results in a remote code execution with the highest (SYSTEM) privileges in the target Windows operating system

Invoking remote RPC function
NetrSendMessage()

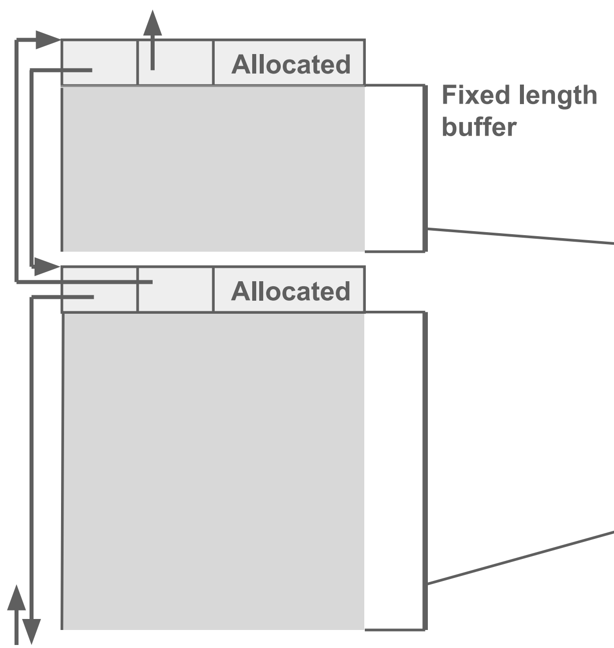
IDL specification

```
error_status_t
NetrSendMessage(
  [in,ref,string] char *_1,
  [in,ref,string] char *_2,
  [in,ref,string] char *_3
);
```

Invoking remote RPC function
NetrSendMessage()

Jumping to specified memory location

Heap blocks



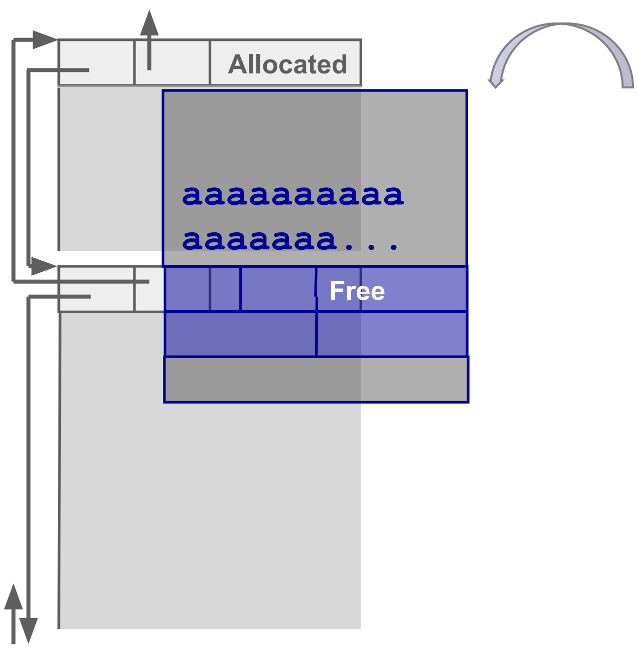
pseudocode

before

```
char *alert_buf_ptr;int alert_len;
NetrSendMessage(char *a1,char *a2,char *a3){
    Msglogsbm(char *a1,char *a2,char *a3){
      alert_buf_ptr=LocalAlloc(0x40,0x11ca);
      Msghdrprint(a1,a2);
      Msgtxtprint(char *a3,int a3len){
        char *ptr=LocalAlloc(2*a3len+1);
        memcpy(alert_buf_ptr+alert_len,a3,a3len);
        LocalFree(ptr);
      }
      MsgOutputMsg(alert_len,alert_buf_ptr){
        RtlOemStringToUnicodeString(...,alert_buf);
        MsgDisplayQueueAdd(alert_buf_ptr,alert_len){
            LocalAlloc(0x40,alert_len);
        }
        RtlFreeUnicodeString(...,alert_buf);
      }
    }
}
```

Jumping to specified memory location

Block header after buffer overflow

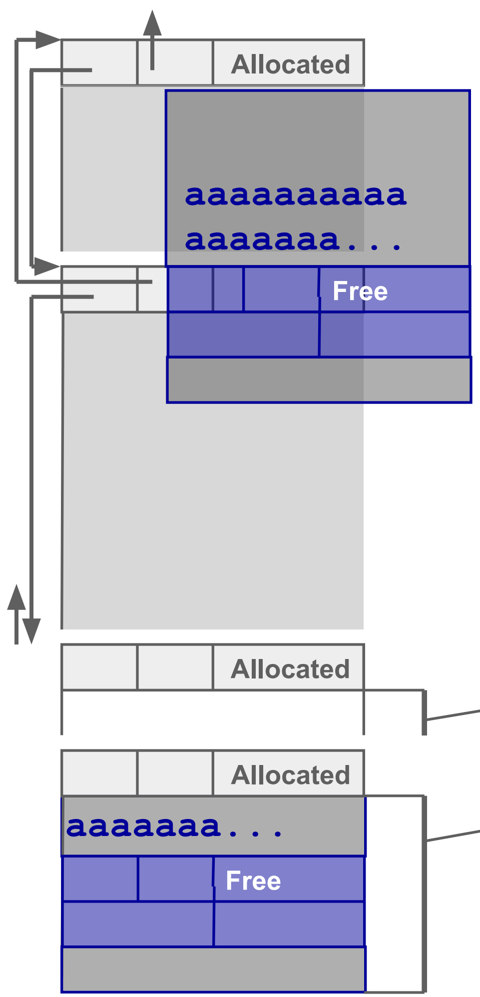


pseudocode

```
char *alert_buf_ptr;int alert_len;
NetrSendMessage(char *a1,char *a2,char *a3){
    Msglogsbm(char *a1,char *a2,char *a3){
       alert_buf_ptr=LocalAlloc(0x40,0x11ca);
       Msghdrprint(a1,a2);
       Msgtxtprint(char *a3,int a3len){
          char *ptr=LocalAlloc(2*a3len+1);
          memcpy(alert_buf_ptr+alert_len,a3,a3len);
after →   LocalFree(ptr);
       }
       MsgOutputMsg(alert_len,alert_buf_ptr){
          RtlOemStringToUnicodeString(...,alert_buf);
          MsgDisplayQueueAdd(alert_buf_ptr,alert_len){
             LocalAlloc(0x40,alert_len);
          }
          RtlFreeUnicodeString(...,alert_buf);
       }
    }
}
```

Jumping to specified memory location

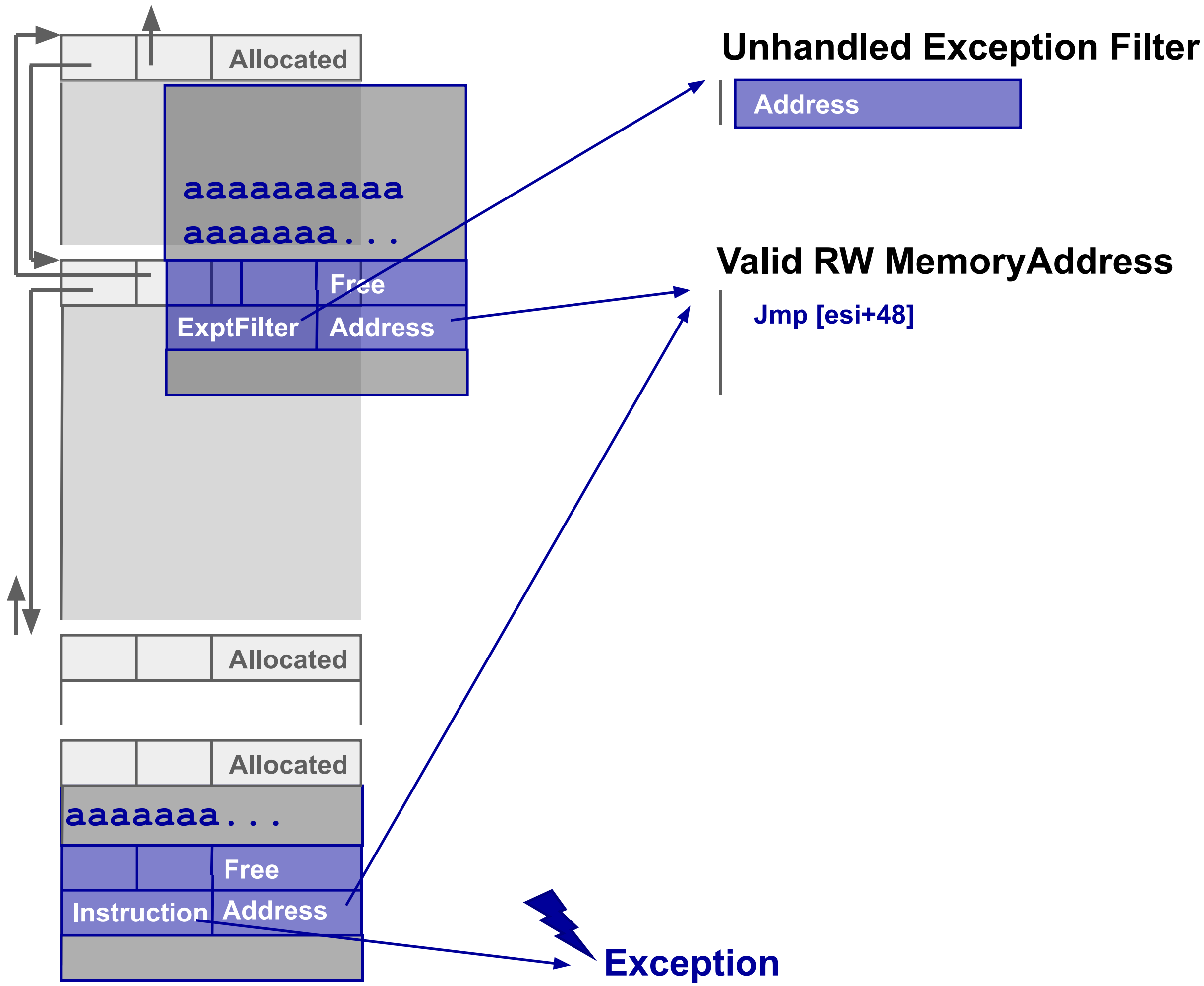
Alloc() and Free() operations



pseudocode

```
char *alert_buf_ptr;int alert_len;
NetrSendMessage(char *a1,char *a2,char *a3){
    Msglogsbm(char *a1,char *a2,char *a3){
        alert_buf_ptr=LocalAlloc(0x40,0x11ca);
        Msghdrprint(a1,a2);
        Msgtxtprint(char *a3,int a3len){
            char *ptr=LocalAlloc(2*a3len+1);
            memcpy(alert_buf_ptr+alert_len,a3,a3len);
            LocalFree(ptr);
        }
        MsgOutputMsg(alert_len,alert_buf_ptr){
            RtlOemStringToUnicodeString(...,alert_buf);
            MsgDisplayQueueAdd(alert_buf_ptr,alert_len){
                LocalAlloc(0x40,alert_len);
            }
before →    RtlFreeUnicodeString(...,alert_buf);
        }
    }
}
```

Jumping to specified memory location
Concatenation of free blocks



Avoiding process crash

Roll back on SEH and fixing the Heap

- The same method as for resuming svchost process state may be used for a process that was a target of Stack and Heap buffer overflow
- Before resuming the process all corrupted Heap structures must be fixed and all used Heap block headers must have appropriate sizes and control flags
- Free block lists must contain only pointers to valid free blocks
- The original pointer to unhandled exception handler must be restored
- In order to resume the process a Divide by Zero exception is triggered and exception handler performs stack unwind operation, restores registers' contents and resumes process execution

Summary

- RPC mechanism is a great example of complex technological component in the context of security
- Existance of a single vulnerability in such a critical component has a great potential impact on security of a whole system
- A complexity of RPC mechanism is one of the biggest difficulty, which can be however reduced by application of effective reverse engineering tools
- Verification of vulnerability's impact is a complex task and its exploitation requires often a lot of work and time

The Last Stage of Delirium

Research Group

Thank you
for your attention!