MSRPC Fuzzing with SPIKE 2006

Dave Aitel

www.immunityinc.com

X'Con2006

Agenda

- Fuzzing overview
- A quick overview of MSRPC and related protocols
- A history of MSRPC fuzzing and drawbacks to these techniques
- Immunity's focus on MSRPC fuzzing
- Future of MSRPC (and hence, of MSRPC fuzzing)

What is Fuzzing?

- We all make our money by taking small strings and making them big strings
- Fuzzing is doing that in a particular way to the inputs of an application
  - Major benefits: No false positives. All bugs you find by fuzzing are reachable (although not necessarily exploitable)
  - Major detriment: problem is intractably slow

Why fuzz?

- Often it's easier to find a bug from fuzzing than reverse engineering, even given an advisory or binary diff
- Fuzzing can find bugs that are difficult to see in the binary or the source code
- A generalized fuzzer for a bug will tell you if a patch is good enough to cover edge cases or if it has an edge case that is still vulnerable

What is the best way to fuzz?

- Non-fault-injection approach
  - We don't inject data directly to the program's API because this would lead to false positives
    - Bypasses authentication, input validation, etc
- Focus fuzzing on finding exploitable vulnerabilities
  - This is not about QA – we tune to the things we're interested in, namely, integer overflows and buffer overflow

What applications are suitable for fuzzing

- Network exposed applications
  - All of DCE-RPC!
- Closed source applications
- Obscure applications that are unlikely to have been reviewed
- Applications that are difficult to obtain
- Applications that are extremely complex
  - Auditing complex applications costs a lot of money!

Fuzzing Mindset

- There's a certain magic to a good fuzzer since there is no guarantee it will find anything
- Fuzzers can take a very long time
  - Weeks, months, etc
- You have to take a leap of faith when you start developing a fuzzer that you're not just wasting your time
- People look down on fuzzers

Problems with fuzzers

- Tokenization is rarely perfect
- Proprietary extensions are easy to miss
- Problem itself is exponential
- Generally only attractive for a blackhat mindset since only finds a subset of potential bugs
- But does give you a good initial indication of the “stance” of the application

How to build a fuzzer

- Tokenization
- Generation of normal traffic
- Generation of abnormalities
- Detection and analysis of problems
- Analysis of quality of fuzzer

Tokenization

- Process of splitting network protocol into invarients (not fuzzed) and variables (fuzzed)
- Types
  - String, Integer, Sizes, Binary blob
- Typical invarients are header strings, protocol constants, calculated responses from network handshakes, etc
- Over-tokenization will make your fuzzer slow
- Under-tokenization will make your fuzzer not find anything

Generation of normal traffic

- Read and parse RFC's or other human-readable protocol descriptions
  – Generally will waste time by fuzzing non-implemented parts of the protocol
  – Will miss proprietary extensions
- Reverse engineering of protocol
  – Can be done semi-automatically
  – If tool is flexible enough, human input can be invaluable
- Sniffing and statistical analysis
  – Even very dumb replay-and-bit-flipping can find many bugs
- If done poorly, target applications will ignore most of your traffic

Abnormality generation

- Transforming normal traffic into malformed traffic, but in a way that is likely to cause exploitable problems
- Bit-flipping is most simplistic
  – For each bit we send, iterate over sending the opposite
- Changing one part of our traffic may require complementary changes to other parts
  – For example, a content-length check

Fuzzing is not Fault Injection

- In fuzzing you go through ALL the layers, starting with the network layer
- In fault injection, you inject incorrect data directly to an API
- There are many layers you don't know about
- Fuzzing never generates a false positive
- Fault injection drawbacks
  – requires a debugger, which may change program operations
  – Generates false positives

X'con 2006

Brief introduction to SPIKE

- First deployed in 2000, one of the first generalized network protocol fuzzers
  - Greg's Hailstorm is the other one (note: very different from current Hailstorm – previously was a commercial fuzzer for arbitrary protocols)
- Introduced unique “block-based” fuzzing
- Included modules for doing HTTP, FTP, and other protocols
- Written in low-level C (for speed)
- Released under GNU Public License

Block-based fuzzing

- Protocols are mostly composed of the same primitives
- Invarients, blocks, and varients
  - <invarient><size><varient><size><var ient 2>
- For each varient replace it with a fuzz string and then update any sizes needed
- For each varient also prepend and append fuzz strings

Advantages of block-based fuzzing

- Linear way to generate abnormalities
- Gut-feel: Finds “interesting” bugs
- Fuzz-streams are reproducible
- Stays close to original valid stream
- Can easily fuzz protocols tunneled inside other protocols

Other Block-Based fuzzers

- Peach (Python-based fuzzer)
  - Free
- Gleg.net ProtoVer (also Python-based)
  - Commercial

Immunity and MSRPC

- 2000 – SPIKE, dcedump, ifids
- 2002 – CANVAS msrpc.py (with auth, local-pipe, and SMB/DCE Fragmentation support)
- 2003 – MSRPC Auditing class
- 2004 – MOSDEF incorporates lexx.py and yacc.py
- 2005 – unmidl.py, DCEMarshall in CANVAS
- 2006 – SPIKE 2006

SPIKE 2006

- Rewritten in Python as part of CANVAS attack framework
  - Takes advantage of pure-python network protocol libraries including DCE-RPC marshaller!
  - Much easier to extend and use
- Added base-string/integer concept
  - A few selected fuzz-variables are used for EVERY variable in protocol while fuzzing
  - Finds somewhat more hidden bugs
  - Is slower (but computers are faster! :>)

SPIKE's Choice of Fuzz Strings

- We use B's instead of A's because of Window's memory flags – B tends to crash right away if we find a heap overflow
- We prepend each long string with \\ and \\\\ and http://
- We use all strings from length 0 to 2200 to catch off by ones
- We also use a set of special strings known to cause problems in the past

Why fuzz MSRPC Applications?

- Thousands of MSRPC interfaces available on default Microsoft applications
  - Writing Microsoft Windows exploits isn't going out of style any time soon
- Also used by many other vendors who build on MSRPC platform
  - These vendors need to test their own interfaces quickly and easily!
- Samba needs regression testing

Overview of MSRPC

- Originally known as DCE-RPC, a competitor to OncRPC and Corba
  - And shares their security issues
- Used mostly on Microsoft Windows as part of DCOM
  - And hence, quite extensively used
- Also available on commercial Unixes
  - Original SPIKE found bugs in AIX's implementation
- Implemented as part of Samba

Components of MSRPC

- Protocol independent
  - UDP/TCP/HTTP/NETBIOS/SMB/etc
- Data-type independent
  - Marshalling and demarshalling allows for encoding of complex data types (with pointers) as network streams
- Encryption and authentication
  - NTLM, security callbacks, etc
- Endpoint mapper

MSRPC Primitives

- Interface
  - UUID
- Interface Version
  - Major and Minor (such as “1.0”)
- Function number
  - 0 to 100 or so

Free (as in speech) MSRPC tools

- Dcedump (port 135)
  - Get list of available endpoints and interfaces
- Ifids
  - Get list of interfaces on a particular endpoint
- Unmidl.py
  - Generates IDL file from executable or DLL

What is an IDL?

- “Interface description language”
  - Explains what the data types are used, which functions are available, and what arguments those functions take
- Generally IDL files are kept proprietary by vendors
  - This makes generating valid traffic difficult
- Compiled by “Microsoft IDL” tool (midl)

Unmidl tools

- Original was GPLed “muddle” by XXXXX
- Python-based GPLed unmidl.py fixed issues with complex structures, pointers, etc
- Followed by <3com product>
- Followed by <free product>

Example IDL (umpnp)

```
  long  Function_36( [in]  [string] wchar_t * 
element_288,
    [in]  long  element_289,
    [size_is(element_291)] [in]  char 
element_290,
    [in]  long  element_291,
    [size_is(element_293)] [out]  char 
element_292,
    [in]  long  element_293,
    [in]  long  element_294
    );
```

Example 2

```
long  Function_09( [in]  [string] wchar_t *  element_825,
[in] [unique]  [string] wchar_t * element_826,
[in]  [string] wchar_t *  element_827,
[in]  [string] wchar_t *  element_828,
[in]  [string] wchar_t *  element_829,
[in] [unique]  [string] wchar_t * element_830,
[in] [unique]  [string] wchar_t * element_831,
[in] [unique]  [string] wchar_t * element_832,
[in] [unique]  TYPE_6 ** element_833,
[in] [unique]  TYPE_6 ** element_834,
[in]  long  element_835,
[out] [context_handle]  void * element_836
 );

 typedef   struct {
 [size_is(524)] char *element_774;
} TYPE_6;
```

A Brief History of MSRPC Fuzzers

- SPIKE
- Samba SMBTorture
- Others?
  - LSD-PL MSRPC fuzzer + unmidl tool lead to MS03-026?
- SPIKE 2006!

Interlude: VERDE

- Found by early version of SPIKE
- Arbitrary free vulnerability in XXXX service
- Reliably exploited by Nicolas Waisman of Immunity
- Fixed in Windows 2000 SP4
- (brief demo)

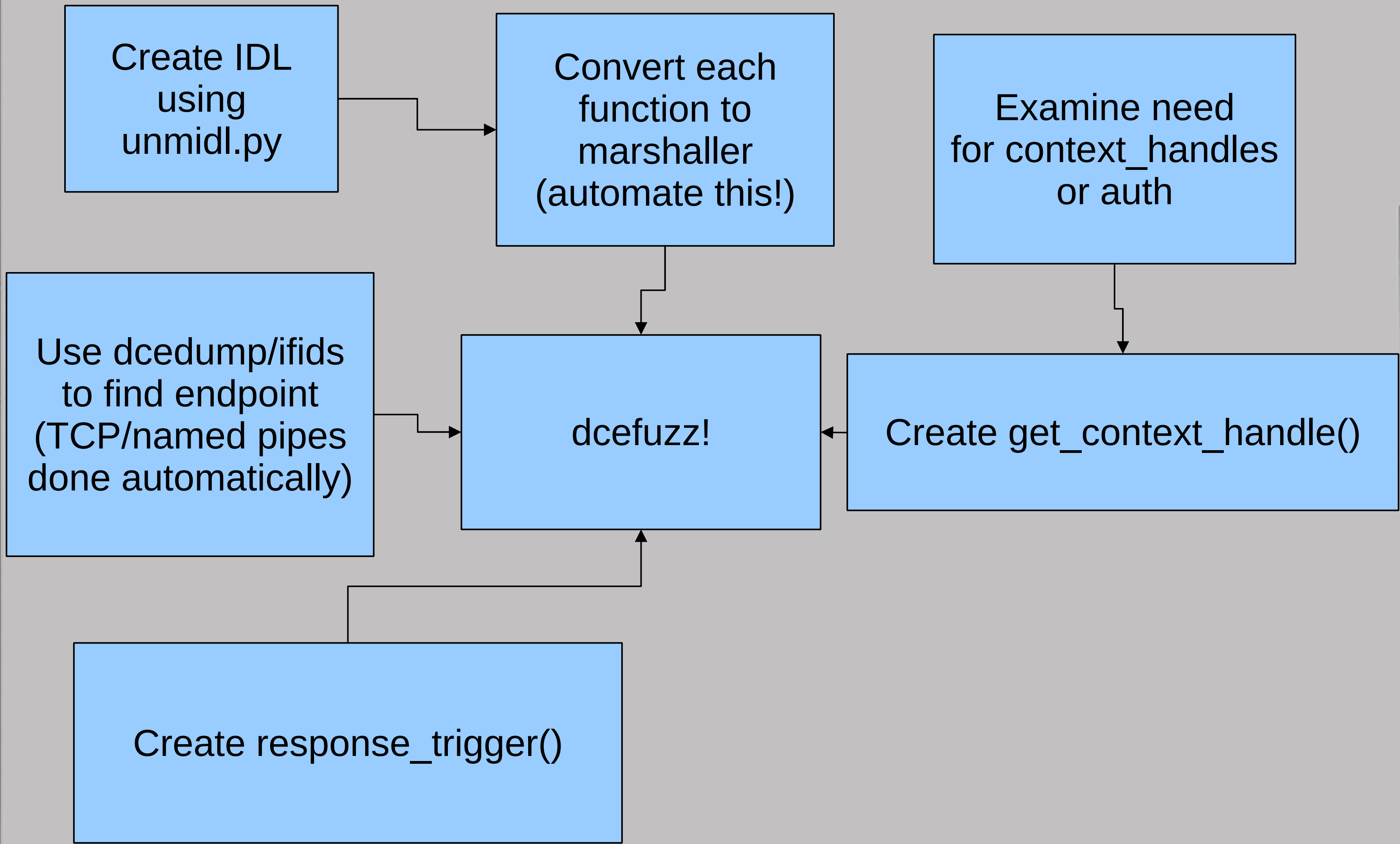
Difficulties in MSRPC fuzzing

- Creating valid protocol stream very difficult
  - Windows 2000 and above check for rigorous protocol compliance – IDL file must be correct!
  - IDL files are not a one-to-one match with demarshalling
- Must include authentication
- Context handles
- Interface may only be reachable locally
  - CANVAS has local named pipe support

Difference in SPIKE 2006 and previous attempts

- unmidl.py improvements
- Working dcemarshaller
  - Complex pointer structures and types can be fuzzed!
- SPIKE offers solution for size_of() arguments
- SPIKE 2006 can fuzz endpoints of almost any type (incl. HTTP, local, etc)
- Response_trigger looks for information leaks, abnormal responses

Fuzzing process



Fuzzing Metrics

- Measuring fuzzers in “number of tests” is like measuring computers in kilograms
- Code coverage is not program state coverage
  - If function_a only crashes when run after function_b, then you can cover both functions and still not reach the crash
    - This is more common than you'd suspect
  - Have to cover code with the right input to find bugs
- Concurrency bugs are hard to “measure”
- Every fuzzer finds different bugs

Fuzzing Metrics (cont)

- Best we can do right now:
  - Does a new fuzzer find all previously known bugs (automatically) and some interesting new bugs?
  - Is it faster and easier to fuzz a protocol than reverse engineer it?
  - Does the fuzzer complete in a reasonable time for the results found?

SPIKE 2006 Results

- Takes (average) under an hour to completely fuzz a given function
- Finds previously known vulnerabilities
- Demos
  - umpnp
  - Exchange DoS
  - ...

Future of SPIKE 2006 and MSRPC Fuzzing

- Automatic fuzzer creation from unmidl and unmidl improvements
- VisualFuzz – Apply SPIKE 2006 techniques via a visual language (like Immunity VisualSploit)
- Use Immunity Debugger
  - To analyze coverage of MSRPC functions
    - Not entire-DLL coverage, but coverage of potential code under the MSRPC Function entry point
  - To create even more correct IDL files

Conclusion

- Fuzzing MSRPC presents interesting problems, which are mostly solved by SPIKE data-structure in a useful way
- Block-based fuzzing scales up to complex protocols
- Questions?