SMASHING THE STACK WITH HYDRA

Pratap Prabhu, Yingbo Song and Sal Stolfo

Columbia University
Intrusion Detection Systems Lab
Overview

• Hydra is a polymorphic shellcode engine for x86.

• Goal: to bypass signature, statistical, and emulator-based IDS.

• Integrates several obfuscation techniques into one engine. Self-cipher, statistical mimicry, fork() code, and more...
INSTRUCTIONS | LOCAL VARIABLE | LOCAL VARIABLE | LOCAL VARIABLE | EIP

Address of Calling function

NOP SLED | PAYLOAD | RETURN ZONE

Overwrites EIP

INSTRUCTIONS | NOP SLED | PAYLOAD | RETURN ZONE

“ret” jumps here
Polymorphic Shellcode

- IDS signatures: \"\x90\x90\x90\x90\" , \"/bin/sh\"
- Use an encoder and cipher the payload with a random key.
- Doesn’t work if the IDS can detect the decoder.
- What about statistical IDS which looks at byte distributions?
- Network emulator, and dynamic disassembly-based IDS?
Hydra Features

- NOP instructions generator.
- Recursive NOP sled.
- Randomized register selection and clearing.
- Randomized multi-layer ciphering.
- Inline junk code/data insertion.
- Multi-partite decoders.

- Multi-gram statistical mimicry.
- Randomized return zone.
- fork()’ing shellcode.
- Time-locked ciphering for anti-emulator and anti-disassembly.
- Alphanumeric encoding.
NOP Sled Obfuscation

• NOP doesn’t have to be \textbackslash x90. ‘A’, ‘B’, ‘C’,..,’Z’ all work

• Hydra contains a “NOP generator” that can build a library of possible NOP instructions.

• Test method:
  – Add code to set up stack/register canary variables.
  – Add a sled built with NOP instruction to be tested.
  – Add validation code to check stack/register variables.
  – Execute.

• Finds NOP equivalent instructions.
NOP Sled Obfuscation

- Not just single-byte NOPS. Multi-byte NOP instructions by way of recursive NOP. (Phrack, CLET)

- Find all 1-byte NOP instructions by brute-force, then find two-byte NOPs where 2\textsuperscript{nd} byte is a one-byte NOP. Repeat.

- Larger NOP instruction recursively contains smaller NOPs. Execution can land anywhere in the instruction.
NOP Sled Obfuscation

- Hydra utilizes two types of NOP instructions.

1. Basic NOP equivalent instructions which can be used to build a sled and safely pass execution into the payload.

2. NOPs which can be safely inserted *between* instructions.

- Second case: “State-safe” NOPs do not contain instructions which modify the stack, registers, control flow, etc.

- 1.9M total NOP equivalent instructions found. 30,000 state-safe NOPs.
Random register operations

• Different synonymous instructions per invocation.

<table>
<thead>
<tr>
<th>Two example ways to clear a register</th>
</tr>
</thead>
<tbody>
<tr>
<td>Method 1:</td>
</tr>
<tr>
<td>mov reg, &lt;key&gt;</td>
</tr>
<tr>
<td>sub reg, &lt;key&gt;</td>
</tr>
<tr>
<td>Method 2:</td>
</tr>
<tr>
<td>push dword &lt;key&gt;</td>
</tr>
<tr>
<td>pop reg</td>
</tr>
<tr>
<td>sub reg, &lt;key&gt;</td>
</tr>
</tbody>
</table>

• Hydra provides a large library of such instructions and a platform to add more.

• For some operations, the key used is randomly generated to further obfuscate the payload.
Multi-partite Decoding

- Hydra generates *non-contiguous* decoders.

- The padded decoder cipher loop is split apart and intermixed with the encoded payload.

- Currently only bi-partite decoding is implemented: half of the decoder instructions are in front of the payload, half after it.

- Decoder instructions jump between each other while decoding the payload.
Multi-Layer Ciphering

- Multiple cipher operations, subsets selected at random per invocation. Very useful technique (ADMmutate, CLET,..)

- Random cipher operations: ROR/ROL, XOR, ADD/SUB, etc...

- Cipher order is random each time.

- A randomly chosen 32-bit key is generated per cipher.

- Six rounds of ciphering by default – user can specify number.
Inline Junk Code Insertion

• Hydra automatically adds space between instructions. Arbitrary data can be inserted:

  [instr 1][junk][instr 2][junk][instr 3][junk][instr 4]

• Amount of data to be inserted can be specified.

• Can insert NOP instructions, anti-disassembly code, random junk, etc. The ciphers will skip these areas during decoding.

• Can also insert certain bytes for statistical mimicry.
Statistical Mimicry

• Statistical IDS – typically work by learning frequencies for normal content then detecting exploits as anomalies.

• Hydra uses machine learning-based techniques to make shellcode mimic normal traffic.

• Learn a statistical model for the distribution of n-grams within legitimate network content.

• Sample from this distribution, and use padding and inline padding (junk insertion) to skew the distribution of shellcode to appear normal.
Randomized Address Zone

- Sequence of repeated target addresses.

- Used to overwrites %ESP on the stack to point to NOP sled.

- An IDS can look for a structural signature such as the existence of NOP instructions and repeated numbers (sled + return zone.)

- Break signatures by adding random offsets to each address element in the return address zone.
Time-Cipher Shellcode

• Emulator IDS? Build stripped down x86 emulator and dynamically execute *ALL* network traffic. Look for self-decryption behavior and/or large basic blocks.

• Solution? Use syscall-based ciphering. Exploit the fact that emulators can’t handle full OS functionality.

• Hydra uses the `time()` syscall. Most significant bits used as key to decode the *main* cipher instructions (ROR, XOR, etc).

• Syscall not handled? Time runs out? Shellcode is decoded incorrectly – no polymorphic behavior is observed.
Time-Cipher Shellcode

- Good for a user-defined period of time. User can adjust the “shell-life” window by the number of bits used.

- Network IDS can’t emulate all possible syscalls.

- Time-ciphered shellcode will pass through the emulators and arrive on the target host where the syscalls can be handled.

- Bypasses some emulator and disassembly based methods, and slows down human reverse engineers.
Forking Shellcode

- Exploit could cause the target process to hang. Not good – could be picked up by an IDS. Graceful recovery (Skyler CanSecWest ’09.)

- Solution: fork()’ing shellcode. Child executes payload, parent *attempts* to recover the exploited process.

- Recovery is hard – correct %EIP is normally lost during exploit.

- Need to know target process address space – relative offset.

- Hydra fork()s your shellcode for you automatically.
Alphanumeric Encoding

• Hydra also incorporates the alpha2 encoder.

• Automatically selects alphanumeric NOPs from the NOP-generator to construct sled. Choice of more than 4000 ASCII instructions.

• Alpha NOPs are inserted in between decoder instructions and shellcode to further obfuscate both content and size.

• Modular nature of the engine allows the Alpha encoding to combine with all of the other options.
Traditional shellcode:

- NOP SLED
- PAYLOAD
- RETURN ZONE

Hydra shellcode:

- Hydra is designed to be modular.
- Shellcode and mimicry bytes intermixed.
- Only ciphers shellcode instructions, mimicry bytes kept in the clear.
THANK YOU DEFCON

Code to be released in the future.

Pratap Prabhu (pvp2105@columbia.edu)
Yingbo Song (yingbo@cs.columbia.edu)
Salvatore Stolfo (sal@cs.columbia.edu)
Statistical Mimicry


Markov chains and Monte-Carlo simulation.

- Hydra accept “training samples” for normal data and learns models for normal traffic.

- Inline-pad shellcode to make it look statistically similar.