Hydra

- Advanced x86 polymorphic engine
- Incorporates existing techniques and introduces new ones in one package
- All but one feature OS-independent

Random register operations

- Different synonymous instructions per invocation.
- 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.

Two ways to clear a register	

Method 1:
 	
mov reg, <key>
sub reg, <key>
	Method 2:
	
	push dword <key>
	pop reg
	sub reg, <key>

Recursive NOP generator

- Traditional shellcode engines use static array of possible NOPs to generate NOP sleds – not very random!
- Hydra uses a built-in “NOP generator” that dynamically builds a library of possible NOP instructions.
- Find all 1-byte NOP by brute-force. Brute-force two-byte NOPs where 2nd byte is another NOP. Repeat. Larger NOP instructions recursively contain smaller NOPs – irrelevant where control flow lands.
- More than 1.9M NOP instructions found!

Recursive NOP generator

- The NOP instructions can also be used in between the decoder instructions; adds variability to size and content of the decoder
- Two types of NOPs– normal NOPs and “state-safe” NOPs
- State-safe NOP library does not contain instructions which modify the environment (stack, registers, flow control)
- Only these have to be used in between instructions, else state is destroyed!

Multi-Layer Ciphering

- Hydra uses randomly select ciphers on the payload.
- Random cipher operations: ror, rol, xor, add, sub, etc…
- Cipher order is random each time. No signature!
- Random 32-bit keys chosen for each operation.
- Six rounds of ciphering by default – can specify arbitrary any rounds.

ASCII Encoder

- Need to send ASCII payload to text based protocols (HTTP) to evade anomaly sensors.

- Hydra picks ASCII NOPs from the NOP-generator to construct the NOP section. Choice of more than 4000 instructions.

- The ASCII NOPs are also inserted in between decoder instructions and shellcode to further obfuscate both content and size.

- Modular nature of the engine allows the ASCII encoding to combine with any/all of the other options.

Bi-partite Decoding

- Signatures for payloads = Pwned!
- But most IDS systems can look for a “decoder”. Cipher loop: xor, ror, shr, shl, *etc*. Static decoders = fail.
- Hydra uses dynamically generated *non-contiguous* decoders! Different instructions each time, different keys, different positions.
- Currently bi-partite decoding: decoders *wrap around* payload. Ultimate goal: tighter integration within payload.

Spectrum Shaping

- Signatures fail so bust out the math.
- The frequency of bytes which correspond to x86 instructions should look different from those of normal traffic, right?
- Wrong! Hydra does alphanumeric encoding – No binary!
- Hydra pads your shellcode with bytes to make it look statistically similar to normal traffic.
- Just give it sample files, it does the training automatically.

Spectrum Shaping

- Hydra learns a 1-byte distribution for the target, then uses Monte Carlos simulation to make your shellcode mimic this distribution.
- Padding at the end is too simple; Hydra automatically spaces out your shellcode instructions inserts the blending bytes in between these instructions.
- Spacing is adjustable.
- Higher-byte mimicry also possible, under development.

Randomized Address Zone

- Sequence of repeated target addresses.
- Overwrites %esp on stack to point to payload.
- Simple IDS signature: NOPs and repeated numbers = sled + return zone.
- Break signatures by adding random offsets to each address in the return zone. Aim for the middle of the NOP sled.

Forking Shellcode

- Successful exploit = target process hangs!   NOT GOOD
- Solution: fork()'ing shellcode. Child executes payload, parent *tries* to recover the exploited process.
- Recovery is hard – correct %eip is normally lost during overflow.
- Need to know target process address space – relative offset.
- Hydra fork()s your shellcode for you automatically!

Time-Cipher Shellcode

- So can't use signatures, can't use statistics, now what?
- Emulators! Build stripped down x86 emulator. Dynamically execute *ALL* network traffic and look for self-decryption.
- Sounds nuts but people have done it!
  - Polychronakis citation
  - Kruegal dynamic disassembly
- Solution? Syscall-based ciphering! Exploit the fact that emulators can't handle full OS features.

Time-Cipher Shellcode

- Cipher your shellcode with special key that can only be recovered when executing with a real OS.
- Can’t carry the key, that defeats the purpose.
- Need the key to be recoverable from the target.
- Can’t be static.
- Solution: the time() syscall! Use the most significant bytes of result as the key: time-locked shellcode.

Time-Cipher Shellcode

- The key is used to decipher the primary cipher instructions in the main loop body.
- If proper key isn't recovered then main cipher loop doesn't execute correctly – illegal instructions. Payload remains encrypted and undetected by the emulator.
- Cipher chaining – with *time* as the initialization vector.
- Can set a "shell-life" for the code: good for only a short period of time.

Conclusion

- Hydra is a new shellcode polymorphism engine designed to foil an array of known IDS methods.

- Why? Because understanding the problem is half the solution.

- Still under development mostly. For future updates check:
  - Pratap Prahbu: pvp2105@columbia.edu
  - Yingbo Song: yingbo@cs.columbia.edu

- Columbia University Intrusion Detection Systems Lab:
  - http://www.cs.columbia.edu/ids