WEBINAR SUMMARY

October 14, 2021

# Arm’d and Dangerous

**Patrick Wardle,** Founder, Objective-See
**Gary Davis,** Chief Evangelist, Intrusion

## KEY TAKEAWAYS

- Organizations must shift from passive, perimeter-based malware defenses to solutions that meet cybercriminals where they are.
- As Macs have become more popular in enterprise settings, Mac malware has exploded.
- M1 malware contains arm64 code and a universal binary with code for multiple architectures.
- To reverse engineer M1 malware, analysts must understand arm64 basics.
- Since similar anti-analysis logic is used across malware, common approaches can be used to bypass it.
- Intrusion is changing the cybersecurity game with solutions based on bi-directional zero trust.

in partnership with

## OVERVIEW

Apple's new M1 systems (also known as Apple Silicon) offer myriad benefits for macOS users. It is no surprise that enterprise adoption of Mac products is on the rise. macOS and Apple's new M1 system on a chip (SOC) have become attractive targets for malware authors. Before analyzing this malicious code, security analysts must first master topics like reversing arm64 code and bypassing anti-analysis logic. Given that a small percentage of cyberattacks have known signatures, IT teams must also consider new approaches to cybersecurity like continuous monitoring of incoming and outgoing network traffic.

## CONTEXT

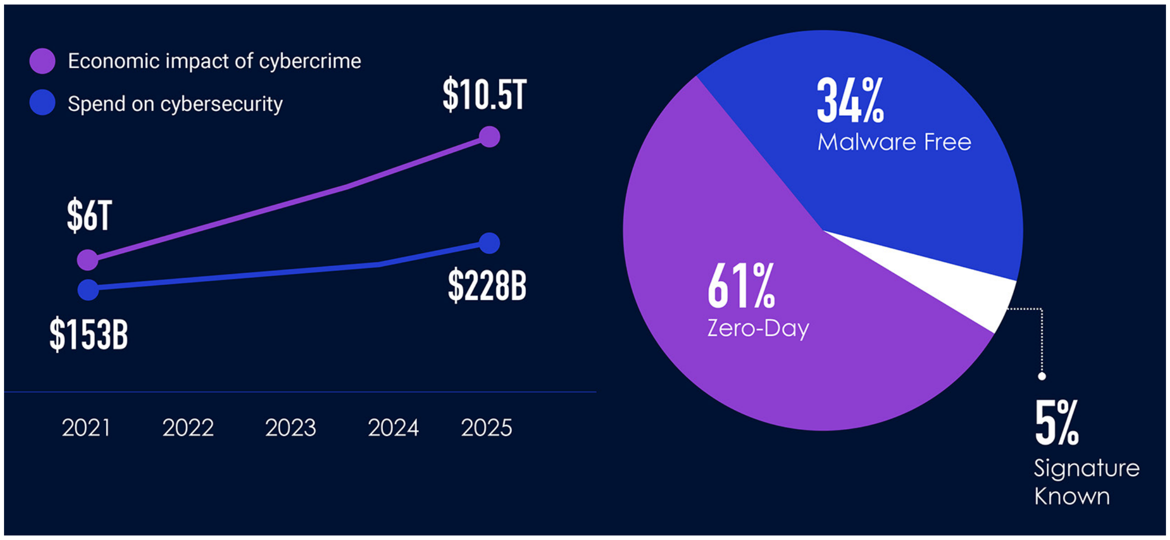
Patrick Wardle discussed how to analyze the anti-analysis logic of M1 malware. Gary Davis discussed the threat landscape and how Intrusion's bi-directional zero trust solutions can protect macOS systems from cybercriminals.

## KEY TAKEAWAYS

### Organizations must shift from passive, perimeter-based malware defenses to solutions that meet cybercriminals where they are.

While IT teams are spending more to address cyberattacks, it feels like a losing battle. Many organizations depend on signature-based cybersecurity products. However, most successful attacks today come from malware-free or zero day types of techniques. Malware with a known signature represents only 5% of successful attacks.

**Figure 1: The Current Cybersecurity Landscape**



Organizations should assume that they are already infected with malware. In addition, Macs are just as vulnerable to cyberattacks as Intel Windows-based machines. In response, IT teams must take four steps to shift away from passive, perimeter-based defenses that look for known malware:

1. Look at both inbound and outbound network traffic. Sampling a portion of the traffic isn't enough. In addition, the network traffic leaving the enterprise is just as important as the incoming traffic.
2. Turn to AI. Both cybersecurity companies and cybercriminals are leveraging AI and machine learning. AI technologies are crucial for analyzing large volumes of data.

3. **Deploy real-time detection technologies.** On average, it takes companies two days to figure out that they are under attack. That is too long.
4. **Rely on large volumes of data.** The more data that is available, the more intelligent AI becomes and the more effective it is at combatting cyber adversaries.

> Signature-only anti-malware products aren't enough to stop contemporary attackers. We need to broaden our approach and rethink how we stop cybercrime. The adversary is much more sophisticated. They are well-funded, well-organized, and more brazen.
>
> *Gary Davis, Intrusion*

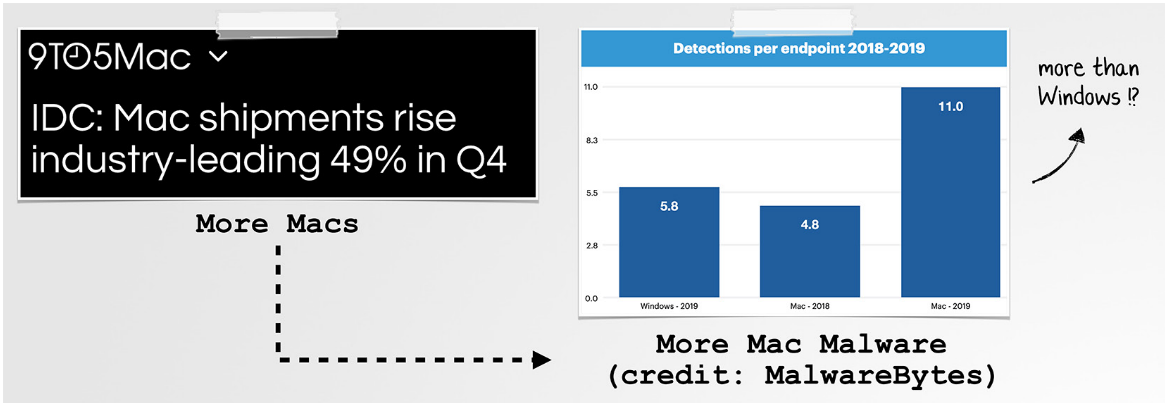
## As Macs have become more popular in enterprise settings, Mac malware has exploded.

Apple's new M1 system on a chip (SOC) is a primary contributor to surging macOS adoption. M1 is also called Apple Silicon. It is an arm-based SOC which means that multiple technologies, like the CPU, the GPU, and DRAM, are combined onto a single chip. This architecture is one reason why the M1 chip is so efficient and performant. Malware compiled to run natively on this chip uses the arm64 instruction set.

Unfortunately, the emergence of M1 malware was inevitable. Any code compiled to run natively on Apple's new chip will run faster and won't be subjected to translation issues associated with Rosetta, Apple's translation technology.

Testing of malware compiled for both Intel and arm64 platforms found about a 10% decrease in the detection of malware on the arm64 sample, even though the malware was identical. This is significant because some signatures are architecture-specific and signature-based cybersecurity solutions may miss known malware, once it is recompiled to run natively on Apple.

**Figure 2: macOS Adoption and the Growth of macOS Malware**

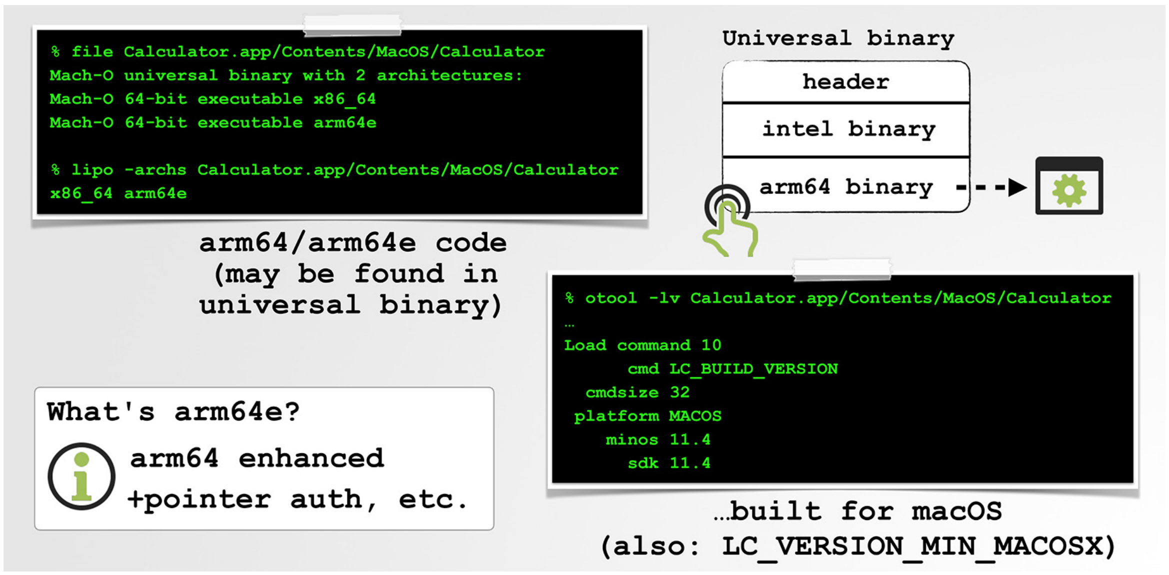


## M1 malware contains arm64 code and a universal binary with code for multiple architectures.

A binary compiled to run natively on Apple Silicon contains arm64 code and within it, there is a universal binary with code for multiple architectures. Mac malware authors distribute macOS binaries in this file format so they can run their code natively on both Intel and arm-based systems.

A file command can be used to look at the architectures and the O tool command can be used to examine load commands within the binaries. This will reveal whether they have been compiled specifically for macOS.

**Figure 3: How to Identify M1 Code**



```
% file Calculator.app/Contents/MacOS/Calculator
Mach-O universal binary with 2 architectures:
Mach-O 64-bit executable x86_64
Mach-O 64-bit executable arm64e

% lipo -archs Calculator.app/Contents/MacOS/Calculator
x86_64 arm64e
```

```
% otool -lv Calculator.app/Contents/MacOS/Calculator
…
Load command 10
      cmd LC_BUILD_VERSION
  cmdsize 32
 platform MACOS
    minos 11.4
      sdk 11.4
```

Earlier this year, Patrick Wardle discovered the first candidate sample of malicious code that was natively compiled to run on Apple M1 systems, called GoSearch22. His analysis revealed that it was a variant of the Pierrot malware family. Since GoSearch22 was identified, there has been a massive increase in arm64 M1 malware. Unfortunately in some cases, Apple has inadvertently notarized these binaries.

## To reverse engineer M1 malware, analysts must understand arm64 basics.

Patrick Wardle reviewed several foundational concepts related to the arm64 instruction set:

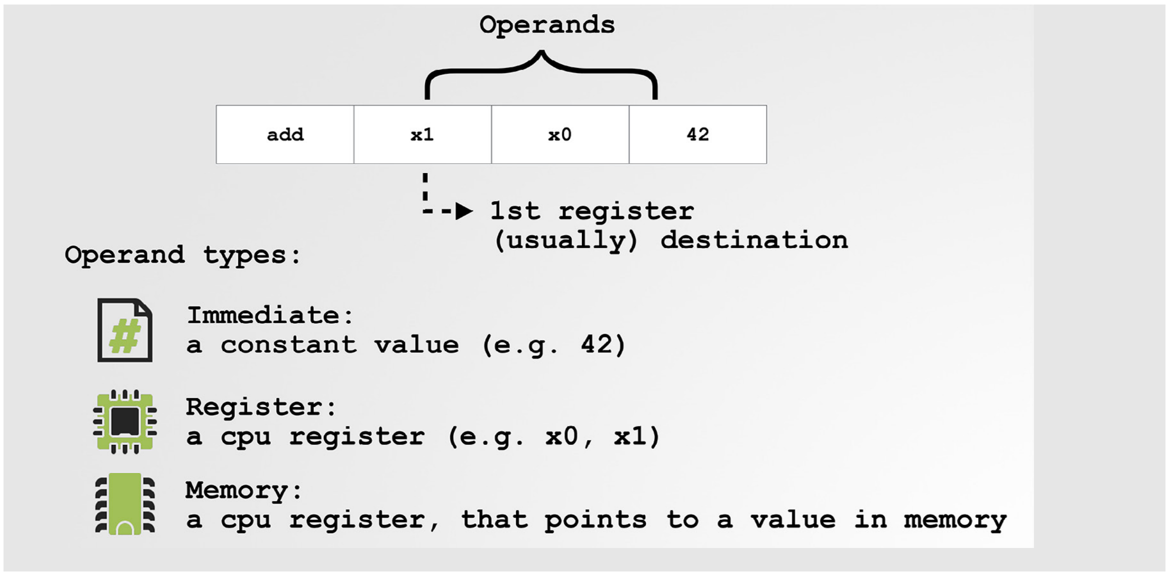
- Registers. Arm64 has 31 64-bit, general purpose registers named x0 to x30. The lower 32 bits of any of these registers can be referred to with the w prefix. There are also dedicated registers. For example, sp is the stack pointer and PC is the program counter. The virtual register is always set to zero. In addition, arm64 has the PSTATE entity that contains condition flags.

  During malware analysis, analysts can gain a fairly comprehensive understanding of a code sample by studying the API calls it makes and the values of the arguments passed to it. In the context of an API call, certain rules define how registers are used. With that understanding, it is possible to identify which register will always hold the first argument and which register will always hold the return value of the function when a function call is made.

  In the context of arm64, the first eight arguments are passed in x0 through x7. The return value of the function will be found in x0, or if it is a 128-bit value, it will be found in x0 and x1. The return address, which is where the function returns to, can be found in the x30 or (lr) register and the x29 register. The frame pointer register is used for stack frames within the context of the function.

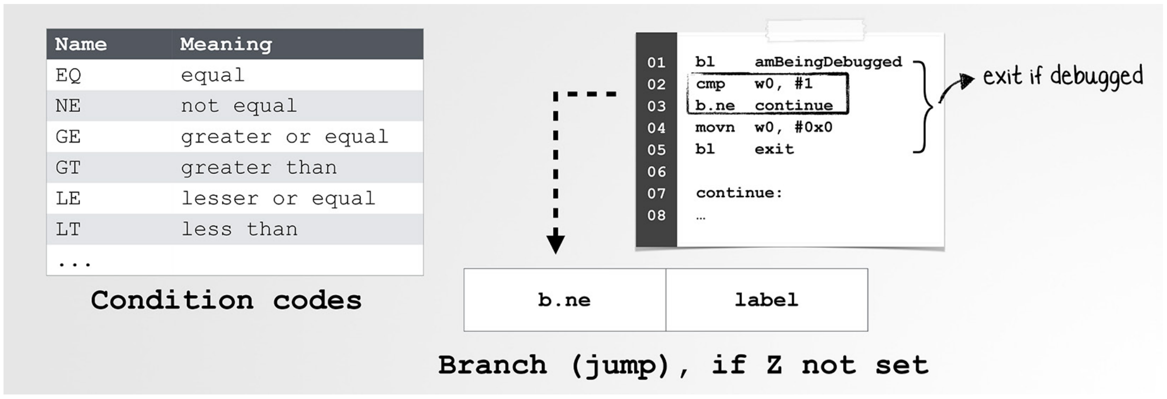
- Instructions. These tell the CPU what to do. Instructions start with mnemonics which are human-readable abbreviations for the instructions that they perform. The remaining part of the instruction is made up of operands. There are three types: immediate, register, and memory. The first operand is almost always the destination register.

Figure 4: Arm64 Instruction Operands


Operands
add
x1
x0
42
1st register (usually) destination
Operand types:
Immediate: a constant value (e.g. 42)
Register: a cpu register (e.g. x0, x1)
Memory: a cpu register, that points to a value in memory

- **Memory access model.** Arm64's load-and-store architecture separates instructions into those that can access memory and those that cannot. The ldr or load register instruction loads information from memory. The str or store register instruction stores a value from the register into the specified memory address. This instruction is unique since the first operand is the source register and the second is the destination.
- **Comparisons and conditions.** Malware often performs checks via comparisons and then takes actions. This behavior can impact malware analysis. Compare instructions perform a comparison between two operands and update the flag in the PSTATE entity. Once condition flags have been set, subsequent instructions act on these flags using condition codes.

Figure 5: Condition Codes



| Name | Meaning |
|---|---|
| EQ | equal |
| NE | not equal |
| GE | greater or equal |
| GT | greater than |
| LE | lesser or equal |
| LT | less than |
| ... | |

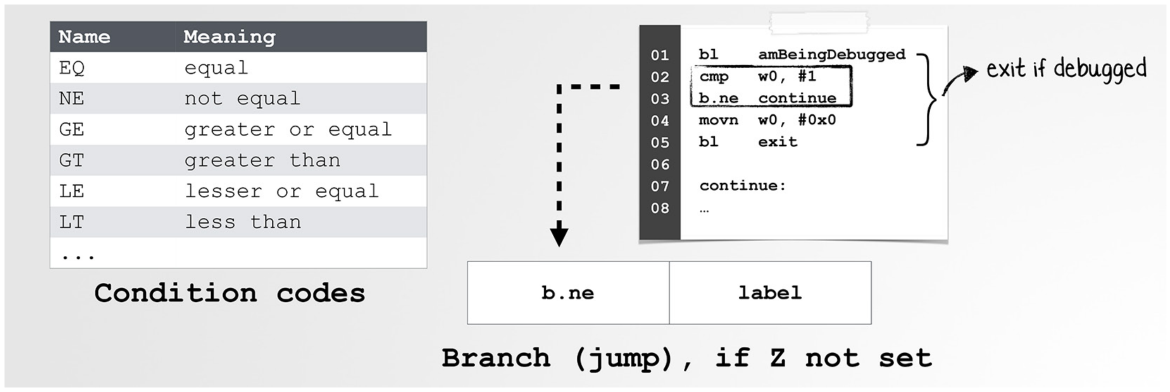
```
bl     amBeingDebugged
cmp    w0, #1
b.ne   continue
movn   w0, #0x0
bl     exit

continue:
…
```

- **Branching.** This is how the arm64 instruction set can alter the control flow of a program. There are three types of branches: unconditional, conditional, and branch and store.

By leveraging a decompiler and dynamic analysis tools, often a fundamental understanding of arm64 will suffice for analyzing malware code. You only need to focus on code arounds and function calls, rather than understanding every assembly instruction.

*Patrick Wardle, Objective-See*

**Figure 6: Branches**



| Name | Meaning |
|---|---|
| EQ | equal |
| NE | not equal |
| GE | greater or equal |
| GT | greater than |
| LE | lesser or equal |
| LT | less than |
| ... | |

```
bl    amBeingDebugged
cmp   w0, #1
b.ne  continue
movn  w0, #0x0
bl    exit

continue:
…
```

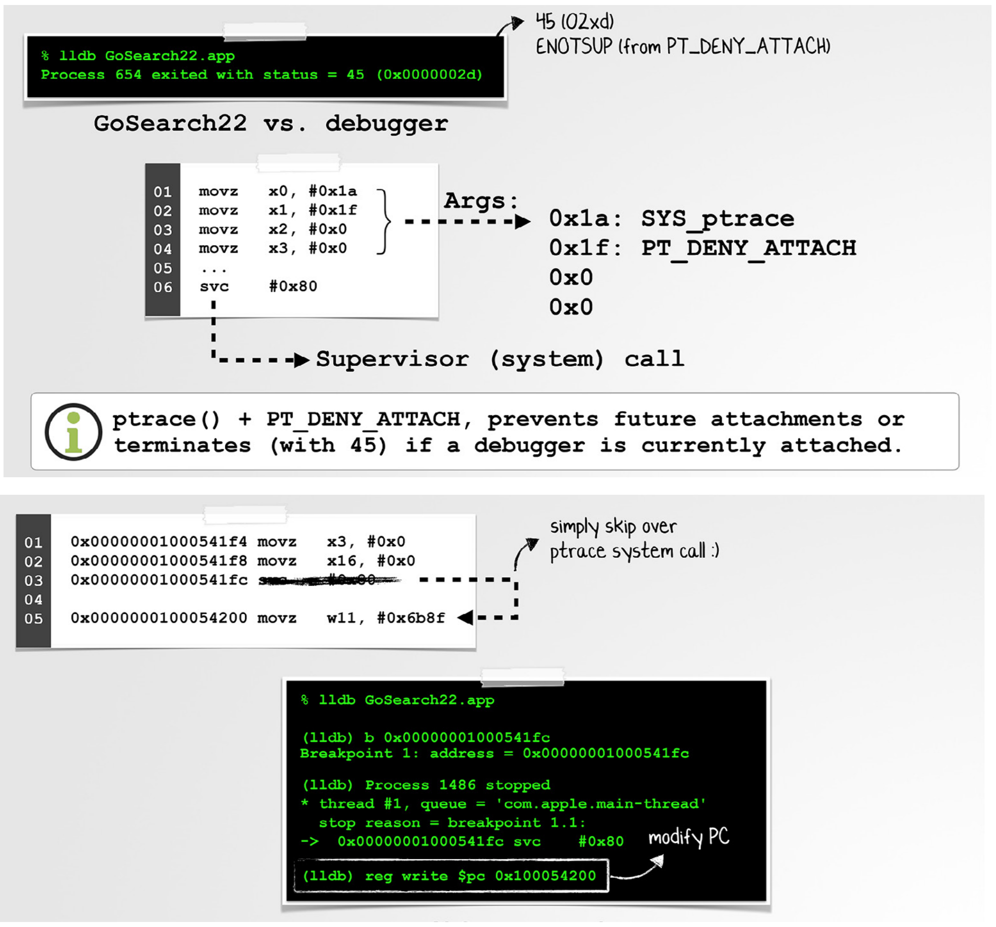
## Since similar anti-analysis logic is used across malware, common approaches can be used to bypass it.

Anti-analysis logic is added to malware to thwart malware analysts. Before a malware sample can be analyzed, it is crucial to identify and bypass the anti-analysis mechanisms. Examples of anti-analysis logic found in the GoSearch22 malware include code to thwart dynamic analysis and debugging tools, as well as code to obfuscate static analysis like garbage instructions and spurious function calls.

In the context of the GoSearch22 malware, Patrick Wardle reviewed three types of anti-analysis logic and techniques to bypass them, debug the malware, and conduct a comprehensive analysis:

1. Debugger detection via ptrace/PT_DENY_ATTACH. The disassembled arm64 instructions for the malware include five instructions which tell the operating system to exit if a debugger is detected. To bypass this logic, the analyst can simply set a breakpoint. Once the breakpoint is set, the reg write command can be used to modify the instruction pointer PC to point to the next instruction. As a result, the problematic supervisor call is never made, thwarting and bypassing the anti-debugging logic.
2. System integrity protection (sip) status detection. System integrity protection (sip) protects binaries from debugging. Analysts typically turn this functionality off when analyzing malware. As a result, malware authors often assume that if malware is run on a system without sip enabled, the code is probably being analyzed and it should exit prematurely. Using the debugger, it is possible to see the value of the register, identify the functions that the malware is about to call, and consult Apple documentation for those functions. By querying the object in the debugger, it is possible to detect that it is going to check the status of system integrity protection in macOS. Once detected, the analyst can change the instruction pointer to skip over this problematic call.
3. Virtual machine detection. The GoSearch22 malware also executes a large script that looks for artifacts of virtual machines. If the malware detects that it has been virtualized, it will exit prematurely, assuming that most analysis is conducted in a virtualized environment. Once detected, the analyst can once again modify the instruction pointer to skip over the problematic call.

**Figure 7: Bypassing Debugger Detection**



```
% lldb GoSearch22.app
Process 654 exited with status = 45 (0x0000002d)
```

```
movz   x0, #0x1a
movz   x1, #0x1f
movz   x2, #0x0
movz   x3, #0x0
...
svc    #0x80
```

ptrace() + PT_DENY_ATTACH, prevents future attachments or terminates (with 45) if a debugger is currently attached.

```
0x00000001000541f4 movz   x3, #0x0
0x00000001000541f8 movz   x16, #0x0
0x00000001000541fc ~~svc    #0x80~~

0x0000000100054200 movz   w11, #0x6b8f
```

```
% lldb GoSearch22.app

(lldb) b 0x00000001000541fc
Breakpoint 1: address = 0x00000001000541fc

(lldb) Process 1486 stopped
* thread #1, queue = 'com.apple.main-thread'
  stop reason = breakpoint 1.1:
->  0x00000001000541fc svc    #0x80

(lldb) reg write $pc 0x100054200
```

**Intrusion is changing the cybersecurity game with solutions based on bi-directional zero trust.**

Instead of looking at signatures to detect malware, Intrusion's cybersecurity products look at IPs. If network traffic is coming from a known good IP, it is allowed. If it's from a bad or unknown IP, it is blocked. Rather than generating scores of alerts for the IT team, bad or suspect traffic is killed immediately. This is a highly effective approach, since virtually every cybercriminal today relies on malware calling home to initiate a successful attack.

Intrusion's Shield solution uses a repository of threat intelligence with approximately 8.5 billion active IPs. The Savant product sits on the wire, monitors the network, and informs Shield whether the traffic is good or bad. Tracecop is Intrusion's threat intelligence cloud that loads the 8.5 billion IPs into memory.

## ADDITIONAL INFORMATION

- **Patrick Wardle's Slides.** The complete slide deck, including code snippets, is available at Speakerdeck.com.
- **Modern Arm Assembly Language Programming.** This book explores the internals of Arm assembly language and how to reverse engineer arm64 code.
- **The Art of Mac Malware.** This open source book by Patrick Wardle can be downloaded for free.

- **Arm64 Assembly Crash Course.** This resource can be found online.
- **How to Read arm64 Assembly Language.** This blog post is available at Scott Wolchok's website.
- **Introduction to Arm Assembly Language Basics.** This tutorial is available at the Azeria Labs website.

## BIOGRAPHIES

### Patrick Wardle

Founder, Objective-See

Patrick Wardle is the founder of Objective-See. Having worked at NASA and the NSA, as well as presenting at countless security conferences, he is intimately familiar with aliens, spies, and talking nerdy. Patrick is passionate about all things related to macOS security and thus spends his days finding Apple 0days, analyzing macOS malware, and writing free open-source security tools to protect Mac users.

### Gary Davis

Chief Evangelist, Intrusion

Gary Davis serves as Intrusion's chief evangelist. Prior to joining the company, he spent over 10 years at Intel Corporation and McAfee, LLC where he served as chief security evangelist. During his tenure he spoke on topics and trends in digital security including the evolving threat landscape, securing the internet of things, the challenges with cryptocurrency, and many more. He has presented at high-profile conferences and events around the world, including CES, Mobile World Congress, SXSW, and the World Knowledge Forum. He has also delivered digital security topics to major educational institutions including Harvard University and Columbia University.

Gary has appeared on broadcast outlets covering breaking cybersecurity news, including CBS News, CNBC, NBC, ABC, FOX News, ESPN, Bloomberg, and WSJ MoneyBeat. He has also been quoted on CNN and in the *New York Times*, the *Wall Street Journal*, *USA Today*, *Money Magazine*, *Forbes*, *Entrepreneur*, *U.S. News and World Report*, and *Time Magazine* to name a few. He has also been featured on radio programs in markets across the country.