A Technique for Secure Vehicle-to-Vehicle Communication

John Launchbury, Dylan McNamee, Lee Pike
Galois, Inc.

Introduction

The recent proposals for Vehicle-to-Vehicle (V2V) communications enable compelling new functionality, but if improperly specified, designed, or implemented, V2V networks could put our nation's vehicles at risk at a massive scale. In a nightmare scenario, unscrupulous actors could disable selected vehicles at selected times, gridlock metropolitan areas at will, or intentionally cause traffic accidents, putting many lives at risk.

News stories are emerging about the vulnerabilities that have been exposed by connecting legacy SCADA systems to the Internet[1]—systems such as dams or even temperature controllers in schools. Similarly, real-world cyber vulnerabilities in modern automotive systems have been demonstrated, including infection through wireless automotive interfaces[2][3].

When semi-autonomous entities with a common vulnerability are interconnected, they are vulnerable to rapid and widespread outbreak of infection. Like a contagious disease, the Morris Internet Worm and Conficker spread through Internet-connected computers that were vulnerable to particular types of attack. Similarly, If V2V systems are widely deployed with cyber vulnerabilities, an attacker may be able to create something like an Internet worm that spreads and replicates through the V2V protocols. The currently known suite of automotive attacks are not self-replicating, in part because the vulnerable cars lack V2V communication capability. But a widely deployed unprotected V2V system could change the landscape and enable attacks with exponential growth phase and widespread effect.

Fortunately, we are still at the design stage of V2V communication. As a nation we do not have to deal with legacy issues of retaining backward compatibility. *Rather we have the opportunity to design and build*

---

[1] See http://www.zdnet.com/blog/security/shodan-search-exposes-insecure-scada-systems/7611 and http://www.infosecurity-magazine.com/view/37005/scariest-search-engine-on-the-internet-just-got-scarier/

[2] Koscher et al., *Experimental Security Analysis of a Modern Automobile*, IEEE Symposium on Security and Privacy, 2010.

[3] http://www.forbes.com/sites/andygreenberg/2013/07/24/hackers-reveal-nasty-new-car-attacks-with-me-behind-the-wheel-video/

V2V communication with security in mind from the outset so we can dramatically reduce the chance of worm-like infections spreading on the V2V network. This paper briefly describes a technique for designing and implementing the interfaces to vehicles that will automatically rule out a whole class of security vulnerabilities. The technology is known, the cost is small.

A Case Study

We start with a recent example. As part of the High Assurance Cyber Military Systems (HACMS) program, DARPA is developing new unpiloted air vehicle (UAV) software. To create a security baseline, the HACMS red team analyzed both open-source and military systems, and found serious security vulnerabilities at the interfaces. The open-source systems had no encryption and no authentication, and perversely, neither did the military vehicles when the mission-profile did not permit the use of their Type I systems, so the red team was able to masquerade as a legitimate base station. They were then able to exploit the permissiveness of the message formats, which allowed messages to be transmitted and received inappropriately—including messages to reboot the system while in flight! They were also able to exploit flaws in the software that processed the input, triggering buffer overflows and gaining deeper control of the system.

That was the baseline assessment. One year later, the same red team attempted to break into the first iteration of the HACMS UAV software. They were unable to do so.

The approach taken in the HACMS UAV software was simple but effective. First, lightweight cryptography was added, ensuring all packets are encrypted and authenticated (providing secrecy, and prevented spoofing, respectively). Second, message formats for flight-specific behaviors were specified precisely, and the software code for input and output of these messages (called *parsers* and *serializers*) were *machine-generated* directly from the specifications. This ensured that the code on the ground stations and the UAVs were consistent with each other. Moreover, once the code-generator was analyzed for correctness, the security of the parser and serializer was automatically assured[4].

With these simple approaches consistently applied, the HACMS UAV software raised the bar on security, leading another DARPA program manager to assess that already the HACMS UAV is likely the "most secure UAV in the world."

[4] http://smaccmpilot.org

Interfaces Introduce Vulnerabilities

Why does protecting interfaces have such a profound effect on security? It turns out that almost all insecurity in computing systems is manifested as (1) the ability to carry out unexpected computation on a target platform by (2) carefully crafted inputs that violate the input assumptions of the design.

Some of the assumptions are visible in the source code. SQL injection attacks are an example of this. Incautious web implementors would ask the user for input such as a name or address, and—making the assumption that the user is only providing a name or address—would paste the input directly into an SQL code template, which they proceed to execute. So long as the user supplies just a name or address, everything is fine. But if a user supplies a piece of active SQL code instead of a name, the web site still pastes it into the code template, and now starts executing the supplied code however damaging it may be. SQL injection attacks are prevented by carefully specifying the set of legal inputs, and then checking that the input strictly conforms to the legal set. Only then is the data used in a query.

In contrast, some assumption are not visible in the source code, but arise from details of how the implementation language executes its programs. Buffer overflow is a classic example of this: the implementation is expecting an input of a certain size, but when the user provides a much bigger input the system naively places the extra input right on top of where is was storing information of what code to execute next! Consequently, carefully crafted overflowing input can allow an attacker to run whatever code they want, right at the heart of the system.

Sometimes the assumptions are in library code. For example, the specific character sequences that are normally used to indicate string endings can cause different systems to behave differently. Such "null terminators" in the middle of a name in an X.509 security certificate can cause some systems to see different names than others[5]. If the input certificate had the name "bank.com[NULL].badguy.com", some systems would see this as a "badguy.com" address, but Internet Explorer and Firefox will see just "bank.com". If one system validates the certificate and then passes it to another who trusts the validation, then a major security vulnerability is opened up.

In order to find and exploit these vulnerabilities, penetration testers (and attackers) use *fuzz testing*[6]. In fuzz testing, the system under test is inundated with random inputs in order to observe any anomalous behavior, like crashing or overuse of memory. Any such behavior is a chink in the armor. The penetrator will then look for ways to turn the crash into an exploitable error, allowing the execution of code well outside the original design parameters. As Matthew Squair puts it:

---

[5] Dan Kaminsky, Len Sassaman, and Meredith Patterson, "PKI Layer Cake: New Collision Attacks against the Global X .509 Infrastructure," Black Hat USA, August 2009

[6] http://en.wikipedia.org/wiki/Fuzz_testing

> *So as the law of unintended consequences dictates we've ended up with … hard-to-parse protocols requiring complex and bug prone parsers, that in turn can become weird machines for carefully crafted input exploits. Further such parsing is mostly ad-hoc and distributed throughout the code making the problem of formally verifying its security almost intractable. The net result of this is that we've ended up in the second great crisis of computing, that of security.*[7]

Secure Parser Generators

Unchecked (or insufficiently checked) input is arguably *the* main vector for cyber attack. Conversely, no matter if the internals of a system have potential vulnerabilities, if attackers can't get unexpected data through, they won't be able to compromise it.

So how can we ensure that all inputs are sufficiently checked?

One of the foundational results of computer science was the development of input grammars, whose properties say precisely how hard it will be to ensure that the input is valid. For the last 40-50 years, programming-language designers have used *context-free* grammars for their input languages, and automated *parser generators* to automatically generate the programming code for that grammar. The code will both act as a *recognizer* (checking any input and rejecting anything that is invalid), and also turn it into a data structure that the rest of the program can use (a *parser*).

Sadly, this standard technique has largely gone unused outside the compiler community, and at great cost to security. Experience has taught us that it is much easier to introduce security errors in handwritten code than in machine-generated code. Even the security protocols in the operating systems of many machines are built on handwritten recognizers. These can lead to direct flaws such as buffer overflow, or indirect flaws such as accepting a larger set of inputs than the protocol specifies. This can set up a difference between sender and receiver that an attacker could exploit, or it could have the pernicious effect of forcing other implementations to have to consciously diverge from the standard in order to interoperate with a popular—but flawed—device.

In contrast, machine generated parsers and serializers can guarantee consistency between sender and receiver. They can also ensure fully sanitized inputs: no values out-of-range, no values of the wrong type, only legal messages, sane routing numbers, etc. This creates a valuable *separation of concerns*: when the system starts to act on the data it has received, it no longer has to worry about sanitization issues. In contrast, with ad hoc approaches the system is continually having to perform some sanitization in the midst of its computations, since there are no guarantees the data was already correctly sanitized.

---

[7] http://criticaluncertainties.com/2014/02/18/starving-the-turing-beast/

Proposal for V2V Interfaces

This leads to the following recommendation for V2V interfaces:

1. All legal inputs shall be specified precisely using a grammar. Inputs shall only represent data, not computation, and all data types shall be unambiguous (i.e., not machine-dependent). Maximum sizes shall be specified to help reduce denial-of-service and overflow attacks.
2. Every input shall be checked to confirm that it conforms to the input specification. Interface messages shall be traceable to mission-critical functionality. Non-required messages should be rejected.
3. Parsers and serializers shall be generated, not hand-written, to ensure they do not themselves introduce any security vulnerabilities. Evidence should be provided that

   parse(serialize(m)) = m, for all messages *m*, and

   parse(i) = REJECT, for all non-valid inputs *i*.
4. Fuzz testing shall be used to demonstrate that implementations are resilient to malicious inputs.
5. A standardized crypto solution such as AES-GCM shall be used to ensure confidentiality, integrity, and the impossibility of reply attacks.

Appendix: Further Exploration

Sassaman et al[8]

This very readable paper starts "computer insecurity has only gotten worse, even after many years of concerted effort. We must be missing some fundamental yet easily applicable insights into why some designs cannot be secured, how to avoid investing in them and re-creating them, and why some result in less insecurity than others. We posit that by treating valid or expected inputs to programs and network protocol stacks as *input languages that must be simple to parse* we can immensely improve security. We posit that the opposite is also true: *a system whose valid or expected inputs cannot be simply parsed cannot in practice be made secure."* The paper goes on to provide many accessible examples of security failures and shows that automatically generated recognizers would have prevented the vulnerabilities.

Complementary to Current V2V Efforts

The approach we advocate in this paper complements current V2V security efforts, which are focused on protocols and cryptographic techniques for securing the wireless traffic between vehicles. If the protocol-handling code has flaws, they could be exploited to bypass or weaken the V2V encryption. The recent "goto fail" vulnerability in MacOS provides an example of this category of weaknesses. An error in

[8] http://www.cs.dartmouth.edu/~sergey/langsec/papers/Sassaman.pdf

one line of code that selects the cryptographic protocol for a session resulted in a vulnerability that could allow a "man in the middle" attack on web sessions labeled as secure by the browser.

Saltzer and Schroeder

In 1975, Saltzer and Schroeder offered design principles for security conscious systems. These principles were formulated in a very different world, but they have stood the test of time. Over the last 40 years, these principles have been ignored to the detriment of security, and "rediscovered" time and time again as people have become more serious about keeping illegitimate users out.

- ***Economy of mechanism:*** Make any design as simple and small as possible. Security holes multiply whenever a system becomes larger or more intricate.
- ***Fail-safe defaults:*** Require positive permission to access features rather than trying to exclude those that should not have access.
- ***Complete mediation:*** Check that every access to a feature is authorized. Checking only once and providing complete freedom of movement once inside opens up many vulnerabilities.
- ***Open design:*** Security through obscurity doesn't work because secrets leak out eventually. Security-critical code benefits from as many people looking at it as possible, both at its design and implementation.
- ***Separation of privilege:*** Require more than one check before granting access to a feature. A single check may fail, or be subverted. The more checks, the harder this should be.
- ***Least privilege:*** Whenever a component has more authority than it needs to get its job done, there is a security vulnerability.
- ***Least common mechanism:*** Minimize the subsystems that are directly shared between multiple components. Especially be careful with shared code or shared variables.
- ***Psychological acceptability:*** Security that's hard to use will get circumvented. Make sure users buy into the security model.

Robustness principle

(Postel's) robustness principle is a popular design guideline for software: "Be conservative in what you do, be liberal in what you accept from others." It says that code that sends commands or data to others should conform completely to the specifications, but code that receives input should accept non-conforming input as long as the meaning is clear. However, Postel's principle should be used with care to avoid introducing security vulnerabilities. RFC 1122 (1989) expanded on the principle by recommending that programmers "assume that the network is filled with malevolent entities that will send in packets

designed to have the worst possible effect". RFC 3117 characterized several deployment problems when applying Postel's principle in the design of a new application protocol, and recommended that programmers provide "explicit consistency checks in a protocol ... even if they impose implementation overhead."

Beyond Context-Free Languages

Sometimes it becomes important to allow interfaces to have dependencies from one part to another, and these can move the grammar out of the *context-free* class. Attribute grammars provide an effective mechanism here. For example, Davidson et al:

> Protocol parsing is an essential step in several networking-related tasks. For instance, parsing network traffic is an essential step for Intrusion Prevention Systems (IPSs). The task of developing parsers for protocols is challenging because network protocols often have features that cannot be expressed in a context-free grammar. We address the problem of parsing protocols by using attribute grammars (AGs), which allow us to factor features that are not context-free and treat them as attributes. We investigate this approach in the context of protocol normalization, which is an essential task in IPSs. Normalizers generated using systematic techniques, such as ours, are more robust and resilient to attacks. Our experience is that such normalizers incur an acceptable level of overhead (approximately 15% in the worst case) and are straightforward to implement.[9]

Acknowledgements

We would like to thank Pat Lincoln from SRI who provided insightful comments on an earlier draft of this white paper.

[9] Drew Davidson, Randy Smith, Nic Doyle, and Somesh Jha, *Protocol Normalization Using Attribute Grammars*, M. Backes and P. Ning (Eds.): ESORICS 2009, LNCS 5789