Controlling Change via Policy Contracts*

Vassilis Prevelakis and Mohammad Hamad,
Institute of Computer and Network Engineering, TU Braunschweig
{prevelakis, mhamad}@ida.ing.tu-bs.de

Abstract

We introduce a flexible policy definition mechanism for determining whether updated software components should be admitted into an operational safety-critical and distributed platform (e.g. car, airplane, space-based platform). In addition the same mechanism can be used to ensure that the component continues to comply with its policy after admission.

1. Introduction

Software updates in vehicles are becoming more complex as the number of ECUs (Electronic Control Units) increases. Safety critical and non-critical applications must coexist, but should not interfere with each other, or threaten the safety of the vehicle and passengers.

Traditionally, software configurations in vehicles combine software from different vendors. So after each software component is released by its developer, they are all integrated into a new software release for a specific platform (e.g. a specific model of a car) and then tested as a whole. This process ensures that all the individual components can work together and function according to specifications. The implication of this, is that software releases are major events and necessitate the transfer of the entire software platform t the vehicle, ruling out incremental updates to individual components. However, there are many instances where a critical update must be rushed through as in the case of Tesla Motors introducing a flash update to reduce the risk of fires in their cars [Bris14]. In such cases an updated component may need to be installed in an existing platform without the benefit of integrated testing.

2. Security Policy Contracts

The ongoing Controlling Concurrent Change project (`ccc.project.org`) aims to ensure that such incremental changes can be applied to complex and distributed safety-critical systems in an automated way. While this problem covers a very wide range of issues (timing analysis, optimization, etc.), our project looks at the security and integrity aspects of such updates.

We address this by creating runtime policy specifications that come with each software component and allow the runtime environment to (a) determine whether this policy is compatible with the platform policy, and (b) to ensure that during its execution, the software component complies with both its own runtime policy and the platform policy. These policy statements form the security dimension of the negotiation that a new component must engage with the platform before it can be admitted into the system. We, therefore, refer to this policy as the Security Policy Contracts (SPCs).

Aspects of the execution of the software component that are currently encoded into the SPCs include resource allocation, communication links, and a code execution profile. In the next paragraphs we describe these parameters and how they can be encoded into SPCs.

* This research has been sponsored by the Deutsche Forschungsgemeinschaft, project PR1449/1-2 "Security and Integrity for Controlling Concurrent Change," and by the H2020 ICT-32-2014 project SHARCS under Grant Agreement number 644571.

2.1 Resource Allocation Policy

A new software component that is admitted into an existing system will need access to various resources such as runtime memory, non-volatile storage (to store its state, and/or to retrieve the state of an earlier version of this component), CPU (for scheduling purposes), access to physical devices such as actuators, or sensors (attached to the ECU) and so on. In some cases this may provide hints to the platform as to which ECU the new software component should be assigned.

2.2 Security policy for communication links

Containment is an important security goal and we believe that strict control over communication channels will prevent a malicious or malfunctioning component from disrupting the operation of other components in the system. Within a given ECU, the runtime environment uses a capability-based mechanism to enforce inter-process communications policy. We use a distributed firewall architecture [Ioan00] to ensure that all communication that go through the internal network are compliant with policy. As a bonus, our system can ensure communications integrity and privacy (although there are few instances where privacy is desired, integrity is extremely important to protect against spoofing attacks) [Prev15].

2.3 Code Execution Policy

The `systrace` facility in OpenBSD controls the execution of a program by looking at the requests this program makes to the operating system. Whenever a program makes a system call, `systrace` checks the type of call (e.g. `open`) and selected arguments (e.g. file name, flags) against the runtime policy file. If the system call itself or one or more of its arguments are not authorized, then the system will not allow the call to go through and take corrective action (e.g. open a substitute file), which may also be included in the policy.

Our runtime environment is microkernel-based so there are few systems calls to monitor and, in any case, the capability-based access control mechanism of the system is quite satisfactory. Instead, we monitor calls to selected system or program libraries. We do this using a variation of the Instruction Set Randomization (ISR) technique we introduced in [Kero10]. Rather that encoding the entire process code space with one key (like ISR) we use separate keys to encrypt the code of each library. Then the user code will have to go through an access gate to ensure that the proper key is loaded into the ISR register before jumping to the library.

An alternative strategy that we also investigate is the use of Control Flow Integrity mechanisms [Chri16] in cooperation with SHARCS (`sharcs-project.eu`), another project that we are involved.

3. Security Policy Examples

For our example let us consider the case where a software component can control two devices, a sensor and a switch and requires some memory for its operation. If the hardware where it is going to run has both a sensor and a switch it needs at least 1000 bytes, and if only one of the devices is present it needs 350 bytes to control only a switch and 700 bytes to collect data from a sensor. For this configuration we would create a policy contract authorizing the platform integrator to use this component. For our example we use the KeyNote policy language specified in RFC 2704.

```
if (
       (sensor == TRUE && switch == TRUE
              && memory >= 1000))
       || (sensor == TRUE && switch == FALSE
              && memory >= 700)
       || (sensor == FALSE && switch == TRUE
              && memory >= 350)
) -> TRUE
```

```
LICENSEE = integrator public key
```

```
AUTHORIZER = designer public key
```

```
signed by = designer private key
```

If the actual ECU where this software is going to run does not have enough memory, or lacks both a sensor and a switch, the policy check will fail and the component will not be admitted.

By having the component designer sign the policy credential we ensure that nobody else can modify the usage conditions and use the software in a way not foreseen by its designer. Assuming the vehicle trusts its integrator (key), we would not need additional credentials to enable this policy. In other circumstances, we may want to add additional policy (for example specifying that only configurations with both the switch and the sensor are acceptable and then add another credential of the form:

```
if ( sensor == TRUE && switch == TRUE
) -> TRUE
LICENSEE = platform public key
AUTHORIZER = integrator public key
signed_by = integrator private key
```

Now the vehicle trusts the platform key, so in order for the software component which is signed by the designer key to be acceptable, it will need both credentials to be true, which is only the case when the component has access to both a sensor and a switch – if it had access to only one of the two, the first credential would be true, but the second would be false.

More information on how this framework may be used to define policy for the communication links between components and various examples on how a software component policy can be refined as we integrate it within ever larger subsystems, may be found in [Prev15].

4. Conclusion

The contribution of this paper is a framework that allows Security Policy Contracts (SPCs) to be specified early in the design process of a software component and to maintain the integrity of this policy throughout the evolution of the component and its integration with the platform. While such policy could be maintained as project metadata and implemented as static files (containing call flow graphs, resource requirements, channel priorities, security parameters, keys, etc.) this is both extremely tedious and error prone. Static configurations also interfere with future upgrades and configuration changes. By expressing the requirements in a policy language and providing the tools to adapt this policy during development and integration we believe that the correct policy will actually be installed in the target platform, thereby, providing improved security for the entire system.

References

[Bris14] Alex Brisbourne, "*Tesla's Over-the-Air Fix: Best Example Yet of the Internet of Things?",* Wired Magazine, Feb. 2014.

[Chri16] Nick Christoulakis, George Christou, Elias Athanasopoulos, Sotiris Ioannidis, "*HCFI: Hardware-enforced Control-Flow Integrity*," Proceedings of the 6th ACM Conference on Data and Applications Security and Privacy (CODASPY), New Orleans, USA, March 2016.

[Ioan00] Ioannidis, S., Keromytis, A.D., Bellovin, S.M. and J.M. Smith, "*Implementing a Distributed Firewall*," Proceedings of Computer and Communications Security (CCS), pp. 190-199, November 2000, Athens, Greece.

[Kero10] Angelos Keromytis, Vassilis Prevelakis, Gaurav S. Kc, Michael E. Locasto, "*On The General Applicability of Instruction-Set Randomization*," IEEE Transactions on Dependable and Secure Computing, **7**(3), July - September 2010.

[Prev15] Prevelakis, V. and Hammad, M.*,"A Policy-Based Communications Architecture for Vehicles*," Proceedings of the 2015 International Conference on Information Systems Security and Privacy, Angers, France, 9-11 February 2015.