2017 Embedded Capture-the-Flag Challenge and Rules:

Secure Firmware Distribution for Automotive Control

1 Challenge Overview

You're part of a team designing the next big evolution in automobiles – a self-driving car. Cars are complex systems and there are huge number of modules that have to work together. You'll be deploying cutting edge algorithms and will be constantly monitoring the system performance in the wild thanks to the onboard cellular connection. If any bugs pop up, or you want to roll out major improvements, you'll use this same connection to program each module with the latest firmware. Done right, this system could save lives, eliminate traffic, and revolutionize transportation.

Photo credit: Andy Greenberg / Wired

However, it's critically important for the safety of the occupants that this system works properly, and given the headlines over the past few years, you have one major concern: security! Can you imagine if someone was able to fly a drone over your car and install new firmware[1]? Or worse, modify your self-driving car over the Internet[2]? Previous MITRE eCTFs have shown that creating a secure device is harder than it may seem. Even with extensive security reviews, it's easy to miss important vulnerabilities. And of course, you're in a hurry to get your product out to market!

What you really need is a way to send firmware updates to your device so that you can add more features (and fix any security problems) after shipping. This functionality is typically implemented as a bootloader – special code that runs every time the device boots. Normally the bootloader will simply turn the execution over to the installed application firmware but if an update needs to happen, the bootloader will handle it by reprogramming the application firmware before handing over execution.

Unfortunately, firmware updating doesn't solve everything and even creates its own set of security concerns because of the added complexity. Possible threats include:

- Competitors might try to read the firmware in the update (or directly from your device) to steal/reverse-engineer your algorithms and other intellectual property.
- Hackers might try to modify your firmware update to insert malicious code that causes the device to malfunction or act as a pivot-point to attack other devices that it connects to.
- Hackers might try to use the update mechanism to install old versions of firmware that have known vulnerabilities.

Your challenge is to design and implement a system to support secure firmware distribution for automotive control. Your system must meet a set of requirements (specified below) and defend against as many attacks as you and the other teams can think of. You must design and implement a working bootloader as well as a set of supporting tools for things such as: generating keys, provisioning bootloaders with those keys, protecting firmware updates, and installing those updates. Once your system is completed, it will be subjected to attacks from the opposing teams, while you get a chance to attack the designs from the other teams. *The purpose of this scenario is to encourage a focus on security for the embedded system and to allow ALL types of attacks.*

[1] http://uk.pcmag.com/philips-hue-connected-bulb/85962/news/should-i-worry-about-my-philips-hue-smart-lights-hacked-by-f

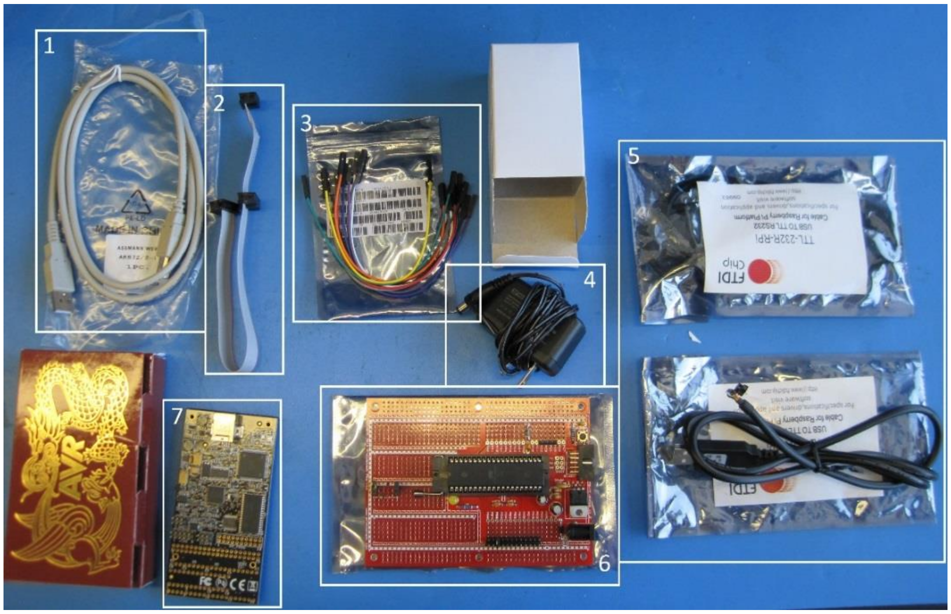
[2] https://www.theguardian.com/technology/2015/jul/21/jeep-owners-urged-update-car-software-hackers-remote-control

2 The Kit

2.1 Hardware

MITRE will provide each team with a set of hardware and peripherals required for the competition. Additional hardware may be purchased by a team if so desired.

Number	Component
1	USB Cable for Programmer
2	ISP Programming Cable
3	Jumper Cables
4	Power Supply for Development Board
5	USB to RS-232 Converters (x2)
6	Development Board
7	Programmer

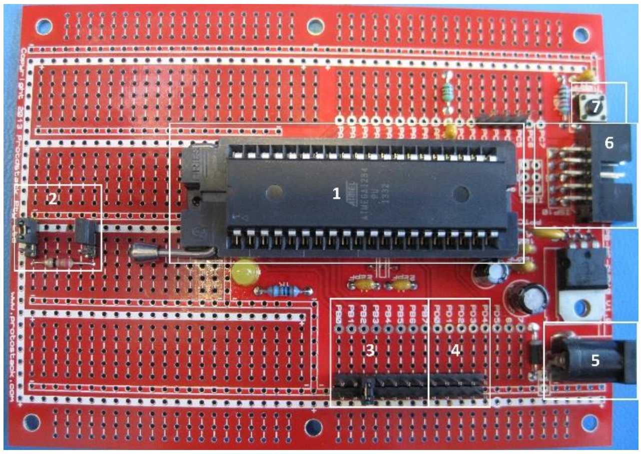


2.1.1 The Chip — ATMega1284P

The microcontroller that we'll be using for this challenge is the ATMega1284P[3]. Check out the Datasheet for the chip[4].

2.1.2 The Development Board — Protostack 40 Pin AVR Development Board

Number	Component
1	AVR Microcontroller
2	Power Side Channel Sense Resistor
3	Port D GPIOs
4	UART Pins
5	Power Connector
6	10-pin ISP Header
7	Reset Button



[3] http://www.atmel.com/devices/ATMEGA1284P.aspx

[4] http://www.atmel.com/Images/Atmel-42719-ATmega1284P_Datasheet.pdf [PDF]

The board[5] is designed to make it easy to build circuits and interface with external hardware. It can be easily modified and instrumented for side-channel attacks, fault injection, or any other analysis you'd like to try. The board has been outfitted with a 40-pin ZIF socket to enable you to swap out microcontrollers.

2.1.3 The Programmer — Atmel AVR Dragon

The AVR Dragon[6] is a programmer/debugger for AVR microcontrollers. It supports the JTAG, ISP, and high-voltage serial programming interfaces.

2.2 Example Bootloader

Building a bootloader from scratch would be a lot of work, so we are providing source code for an example system that meets the functional requirements for the competition. Be warned however, that this example offers no protection from attackers. In fact, many security issues may exist and *persist* if they are not identified and removed.

For more details, refer to the documentation that accompanies the example bootloader.

https://github.com/mitre-cyber-academy/2017-ectf-insecure-example

2.3 Development Environment

Included with the example bootloader we are also providing a Vagrantfile[7] with all the tools necessary to build and test the example bootloader. When you submit your bootloader you will need to include an updated Vagrantfile with all of your dependencies so that we can build and test your bootloader. The submitted Vagrantfile should launch the build VM and build any tools necessary to run or test the design.

Refer to the provided development environment documentation for more information.

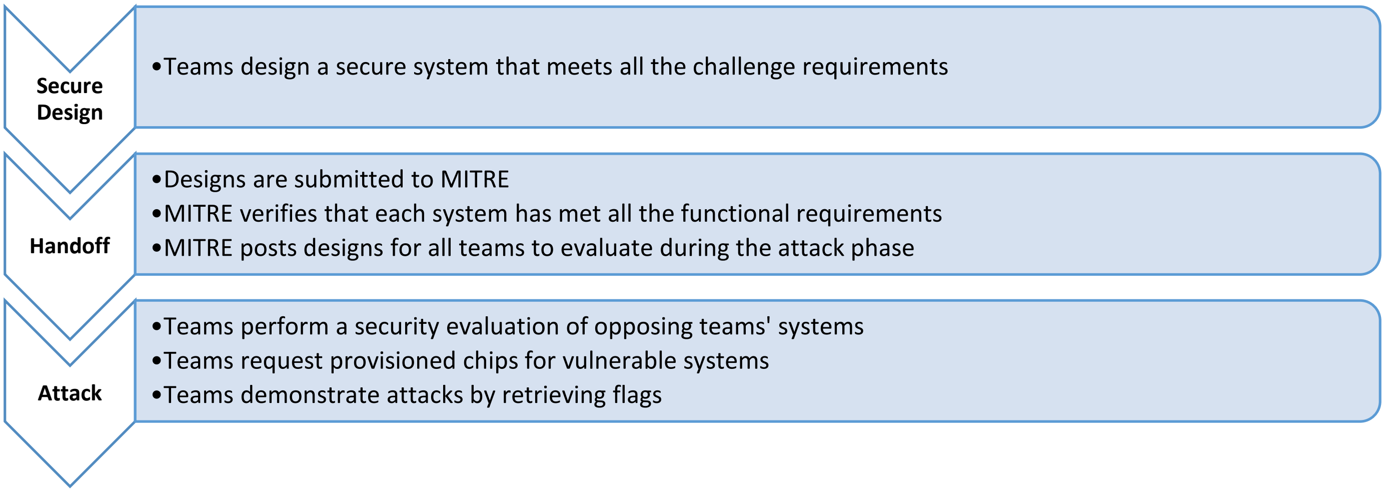
[5] http://www.protostack.com/kits-modules/atmega1284-development-kit

[6] http://www.atmel.com/tools/AVRDRAGON.aspx

[7] https://www.vagrantup.com/about.html

3 Competition Phases

This is an attack-defend capture-the-flag, meaning there are both attacking AND defensive portions. The eCTF is composed of the following phases:



Secure Design

- Teams design a secure system that meets all the challenge requirements

Handoff

- Designs are submitted to MITRE
- MITRE verifies that each system has met all the functional requirements
- MITRE posts designs for all teams to evaluate during the attack phase

Attack

- Teams perform a security evaluation of opposing teams' systems
- Teams request provisioned chips for vulnerable systems
- Teams demonstrate attacks by retrieving flags

Figure 1. eCTF Phases

4 Secure Design Phase

The secure design phase encompasses the design of a secure bootloader and the creation of a total of five (5) support tools to 1) build and provision the bootloader, and 2) secure, install, and debug application firmware. Both the bootloader and the tools have requirements (covering functionality and interfaces) that must be met in order for a design to be considered complete.

4.1 Bootloader Description

Each team must design a secure bootloader that will be loaded onto the ATMega chip. Most of the time, the bootloader will simply launch the application firmware that had already been installed, but it must also support a capability to install new firmware. Other capabilities (e.g. debug logs, verification) must also be supported and are described below. Most of the specifics of these capabilities are left up to the teams.

4.2 Security Goals

4.2.1 Confidentiality

Firmware images should be protected to prevent reverse-engineering or stealing of intellectual property. Anyone with access to the raw firmware could easily extract sensitive information such as proprietary algorithms or security mechanisms. To ensure confidentiality, the firmware should be protected throughout its lifetime (i.e. while at rest, while being transferred to the bootloader, etc.). Only the proper bootloader should be able to extract the raw firmware in order to install it.

4.2.2 Integrity and Authentication

Your bootloader should also contain some mechanism(s) for validating that a given firmware image is from a legitimate source and was not altered during transit. Firmware images that fail these checks should not be installed or loaded; otherwise, nothing would prevent modified/malicious/invalid firmware from being executed on the device. Only protected firmware images that are created in the secure environment (i.e. at the “factory”) should be accepted by the bootloader.

4.3 Functional Requirements

4.3.1 Versioning

Protected firmware images must support a version number that is added by the “Firmware - Bundle and Protect” tool. The purpose of the version number is to prevent an attacker from holding on to firmware images for old versions and installing them later (downgrading) to exploit a known vulnerability in the old version. Once the bootloader installs a new version, it shouldn’t allow the installation of any older version (with one special exception).

1) The bootloader should be initially configured with version number 1
2) Once the bootloader installs a firmware image, it must never install one with a lower version number (i.e. it must only install that same version or higher going forward)
    a) Exception: For debug purposes, a valid firmware with version number 0 is always installable
    b) Installing a firmware image with version number 0 does not reset the version number on the device (i.e. if a firmware with version 5 is on the device, and is then updated with a version 0 firmware, it will not allow firmware with a version number between 1 and 4 to be installed)
3) Version number must support 16-bit numbers

4.3.2 Read-back (Verification)

The bootloader must support a means to provide a technician with access to any region of the flash memory housing the installed firmware. Other regions of flash memory, including the bootloader, may optionally be read. This function, exercised via the “Firmware – Readback / Verification Tool” gives a technician the ability to debug firmware currently loaded on the ATMega, including any flags that are loaded. This feature should only be possible if the technician has access to special security parameters (e.g. a password or a cryptographic key).

4) Bootloader communicates with the host tools over UART1 for update and readback modes.

4.3.3 Boot messages and control of UART0

Boot messages provide feedback to the user during boot by printing the current version of the loaded application firmware and the associated release message. The "release message" is a string that is attached to a protected firmware image by the Protect tool (see section 4.4.3) – think of it as a title or short description for the firmware image.

5) UART0 (configured to 115200 baud) is reserved for status/debugging messages from the bootloader during boot. Once the application firmware is booted, it can take control of it for control and status messages. UART0 does not interact with the host tools.
6) The bootloader must print the release message (on UART0) associated with the currently loaded firmware image before launching that firmware. This indicates that the bootloader has accepted the firmware as valid.

4.4 Host Tools Description and API

The host tools can be split into two groups: those used in a secure setting, such as a factory or at MITRE, and those intended to be used in an insecure environment, such as in the owner’s garage (or in a hacker’s basement). An example of each tool is available in the Example Bootloader package.

Please note that each tool has a specific command line interface which must be followed exactly. These are needed to support our configuration and testing framework to administer the eCTF. All of the tools will be run in the directory in which they reside, so files that are generated by a particular step should be available for use in subsequent steps. All host tools should be built in a specific directory: `~/ectf/host_tools`. All tools should return an exit code of integer 0 (zero) to indicate that the tool has successfully finished running. Any other integer value will be considered an error. Also note that all tools are expected to use a standard baud rate of 115200.

All dependencies of the host tools must be included in the Vagrantfile included with your submission. All Vagrantfiles will be built using the 'vagrant up' command exclusively. If the organizers cannot get your tools working, your submission will not be accepted and you will not be able to start the attack phase (see section 5 for more details on design submission).

4.4.1 Bootloader – Build Tool

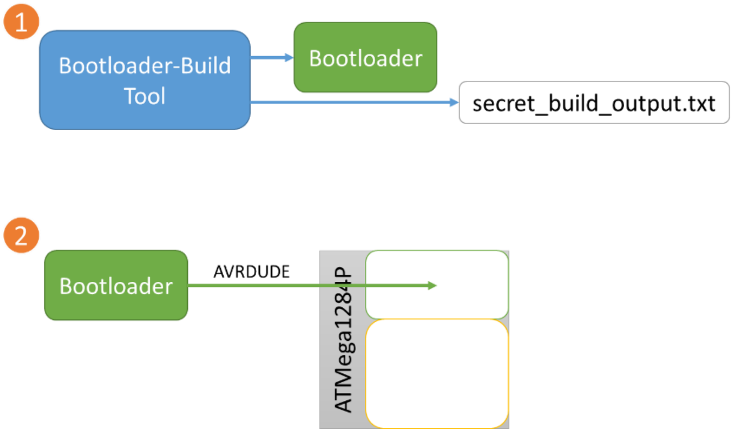


Figure 2: Bootloader - Build Tool

This tool will use a compiler to build a series of .hex files that represent the bootloader. During this process, if cryptographic keys or other security parameters are generated, they should be saved to the file `secret_build_output.txt`.

After this tool has been run, the bootloader .hex files can then be loaded onto the ATMega1284P as the new bootloader via *AVRDUDE* (provided with the kit Vagrantfile). An example of this loading process can be found in the Example Bootloader package[8].

NOTE: The last 512 bytes of EEPROM are reserved for secret values that are used by the application firmware. Any data that is located in this section of memory will be overwritten.

API:

- Cmdline: `./bl_build`
- Writes security parameters (if generated at this stage) to `secret_build_output.txt`
- Compiles bootloader to generate the following files (which will be programmed onto the chip with `avrdude`):
    - `flash.hex` — The contents of flash memory.
    - `eeprom.hex` — The contents of EEPROM.
    - `lfuse.hex` — The contents of the lower fuse bits.
    - `hfuse.hex` — The contents of the upper fuse bits.
    - `efuse.hex` — The contents of the extended fuse bits.

Other .hex files that AVRDUDE accepts via the "-U" command line option can optionally be generated. These additional files should be named "`<memory type>.hex`", such that <memory type> is acceptable as an option to AVRDUDE. For example, `flash.hex` will be provided to AVRDUDE using the command line string "`-U flash:w:flash.hex`", since the memory type is "flash".

[8] https://github.com/mitre-cyber-academy/2017-ectf-insecure-example

4.4.2 Bootloader – Configure Tool

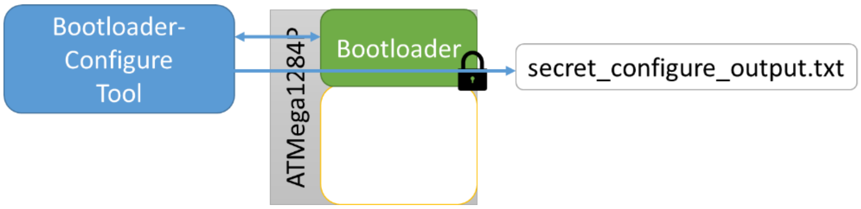


Figure 3: Bootloader - Configure Tool

This tool verifies that the bootloader was programmed successfully. It may also provide (optional) additional configuration of the bootloader that requires active participation of the provisioned bootloader. This verification and configuration (if necessary) occurs over the bootloader control serial interface (UART1). This tool may consume `secret_build_output.txt` for any security details it needs in order to run. All security parameters necessary to protect any aspect of the design must be generated by this point, including the bootloader, future firmware, and readback functionality. All keys and security parameters, generated now or during the build process, must be stored in `secret_configure_output.txt`.

API:

- Cmdline: `./bl_configure --port <serial port>`
- May optionally consume `secret_build_output.txt`
- Writes all security values into `secret_configure_output.txt`

4.4.3 Firmware – Bundle and Protect Tool

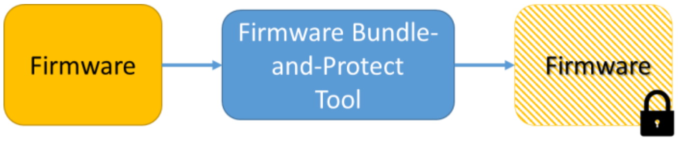


Figure 4: Firmware - Bundle and Protect Tool

This tool protects the sensitive intellectual property in the application firmware from unwanted disclosure. The specific protection used is up to the designers, although it should provide confidentiality as well as integrity and authentication checking (see Security Goals). The largest firmware that this tool must accommodate is 30 kB.

This tool must also add a supplied version number to the protected firmware for compliance with the bootloader requirement of version checking. The version number must support 16-bit numbers. Finally, this tool will add a boot message to the firmware to be displayed during boot. Release messages up to 1 kB must be supported.

API:

- Cmdline: `./fw_protect --infile <unprotected_firmware_filename> --version <version_number> --message <release_message> --outfile <protected_firmware_output_filename>`
- Consumes `secret_configure_output.txt` to protect firmware
- Adds security to firmware.
- Adds version number to firmware.
- Add release message to firmware.

4.4.4 Firmware - Update Tool

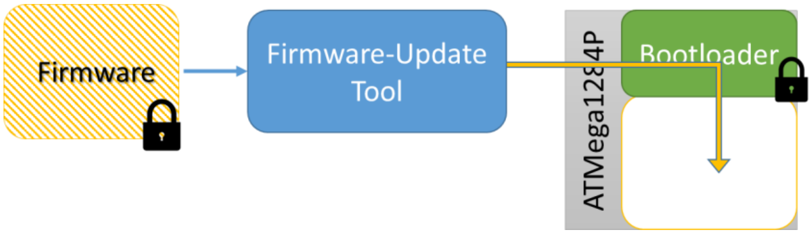


Figure 5: Firmware - Update Tool

This tool installs a new firmware on a target AVR. This tool is used to install the initial firmware version (at the factory) as well as future updates in untrusted environments. Only firmware validated by the bootloader for proper authentication and integrity should be loaded. Additionally, the bootloader should verify version information as discussed in the Functional Requirement section.

API:

- Cmdline: `./fw_update --port <serial port> --firmware <filename_of_protected_firmware>`
- This tool *cannot* make use of the `secret_configure_output.txt` file

4.4.5 Firmware – Readback / Verification Tool

This tool allows for debugging of defective devices that are sent back to the factory. It works by a technician asking for some amount of flash memory starting after a specified memory address. This tool must be able to access any region of the currently installed firmware, but may allow access to other regions of the flash memory as well (including the bootloader itself). The bootloader may demand authorization which the tool can satisfy using the `secret_configure_output.txt` for security keys and parameters, which should be considered available for legitimate use at the factory.

API:

- Cmdline: `./readback --port <serial port> --address <start_address> --num-bytes <number_of_bytes_to_read>`
- Should consume `secret_configure_output.txt` for security parameters

4.5 System Description

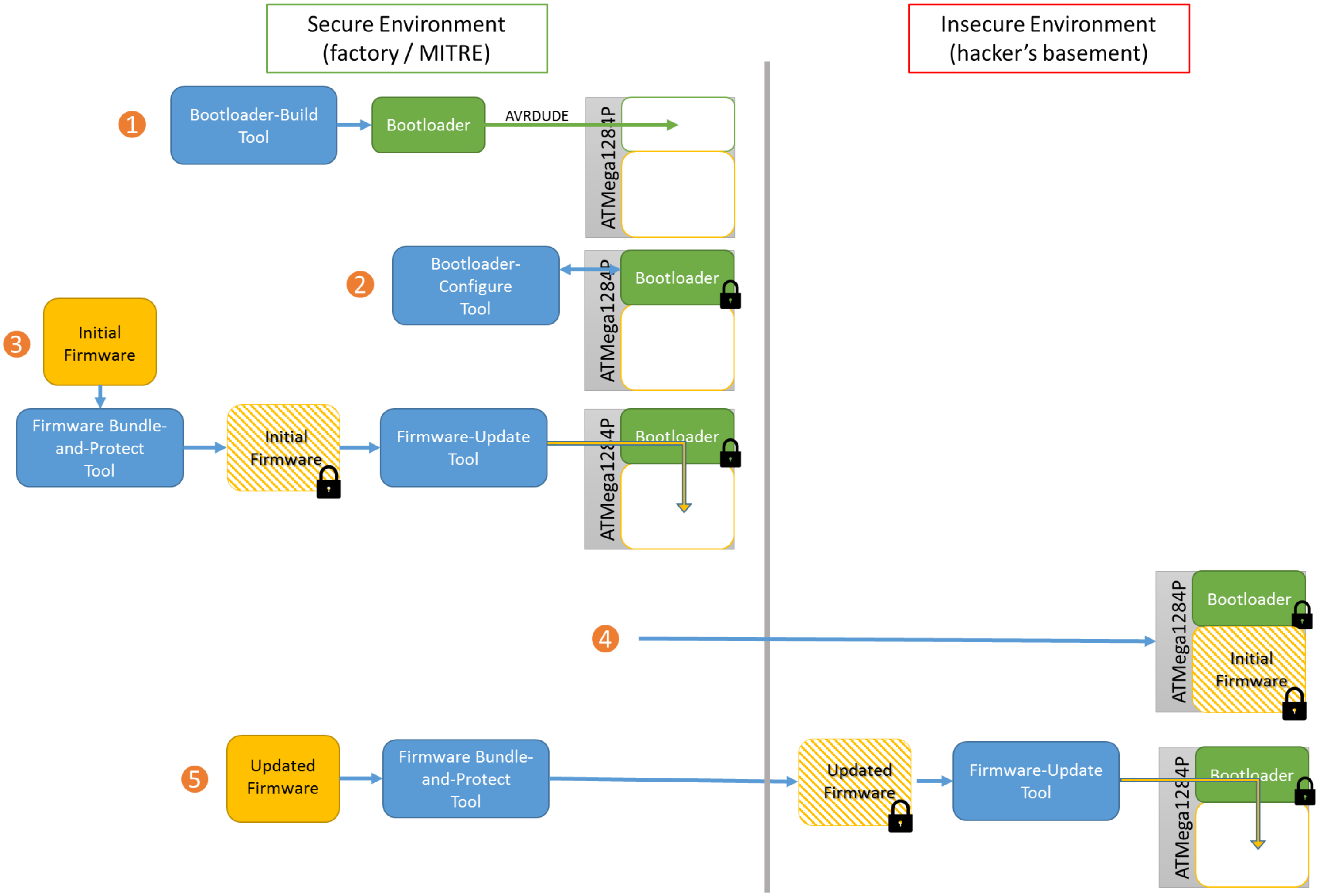


Figure 6: System Use and Timeline

The following steps occur at the secure "factory" at MITRE:

1. The "Bootloader-Build" tool generates the bootloader .hex files, which is then loaded onto the ATMega via AVRDUDE.
2. The "Bootloader-Configure" tool communicates with the bootloader to ensure proper installation and to finalize bootloader security.
3. An initial firmware image (version 2.0) is secured by the "Firmware-Bundle and Protect" tool, after which is then loaded on the ATmega by the "Firmware-Update" tool.
4. The fully provisioned product is shipped to a customer.

At some point in time later, an updated firmware is necessary for all deployed products.

5. The updated firmware is protected by the "Firmware-Bundle and Protect" tool at the factory. The new secured firmware is then accessible by all owners of your product. This is downloaded by a customer, who uses the provided "Firmware-Update" tool to load the new firmware onto his/her ATMega.

Any steps that occur at the factory should be assumed to be safe and any secret values generated there will not be freely available to attackers. Additionally, it is assumed that an attacker will not have access to the microcontroller until after it has been provisioned with the secure bootloader and lock-bits have been set. After the initial provisioning, attackers will have physical access to the microcontroller. Therefore, physical attacks on the microcontroller are fair game. Additionally, attackers will be able to execute firmware updates, at which point they can monitor the communication or monkey with the device during the update process.

A final wrinkle is that a factory technician accidentally released a recording of the readback tool onto the Internet. **Therefore, it should be assumed that all attackers will have access to a recording of the communication from the bootloader to the readback tool.** A cleverly designed update system will minimize the damage caused by this recording.

5 Handoff Phase

After the completion of the Secure Design Phase, each team must submit their design to MITRE. Submissions should include all source code, documentation, and supporting files necessary to build and test the design. All of this should be combined into a single zip file. At the root level of the zip file should be 'Vagrantfile' and a directory named 'host_tools'. The 'host_tools' directory must conform to the host tools requirements described in the official rule documentation. The submission must include source code for the bootloader and tools, as well as any additional documentation for the design. This zip file can be transmitted to us in one of two ways:

1) Email us a URL that points to the zip file, or…
2) Email us the zip file directly (as an attachment)

Submission emails should be addressed to ectf@mitre.org with the subject line: *"eCTF Submission – <Team Name>"*

Where *"<Team Name>"* is the name of your team. There is no required format for the naming of the zip file, although including the school and team name is appreciated.

After receiving a team's design, MITRE will build each tool (if necessary) and validate that the bootloader and all tools meet the functional requirements. Any design that will not build or does not meet these requirements will not be able to progress to the attack phase of the competition. MITRE will contact each team with handoff status within two (2) days after a team's submission, whether it is accepted as functional or not. Teams are able to resubmit updated designs after previous designs were not accepted. Once a team's design has been accepted, they may not submit further designs.

All source code and documentation, as well as the build environment/Vagrantfile, will be provided to other teams during the attack phase to discourage security-by-obscurity, as well as to accelerate attack development and encourage more sophisticated techniques for both sides. Particularly good documentation will be worth extra points at the discretion of the MITRE competition committee – see Section 7.3 for details.

6 Attack Phase

Each design that has been validated during the Handoff Phase is available for attack. For each design, the files listed below will be made available to all attacking teams for "evaluation". These files will be created by running the necessary build tools, but will not contain real flags and the fuse files will be modified such that they will not prevent debugging, memory readback, or writing over the bootloader. Teams should use these files to develop proof-of-concept attacks using your own development hardware.

- Source code for the bootloader and host tools
- Build environment (Vagrantfile)
- Documentation
- Pre-built bootloader and firmware files
    - All .hex files
    - Protected firmware v1
    - `secret_build_output.txt` and `secret_configure_output.txt`

- Capture of readback tool communication accessing a small amount of memory

The recording of the readback tool will include memory that contains the Readback Sniffer flag. This recording will include all of the serial communication as well as timing of the communications to some resolution.

Teams in the attack phase will be given login credentials for the competition scoreboard website. Upon logging in to the scoreboard site, links can be found to download all of the available "evaluation" packages for attack analysis. The scoreboard also accepts the submission of flags for points.

Once a team has developed an attack against a specific design (that they believe will capture a flag) they may request a chip provisioned with that design from MITRE. This provisioned chip will contain the real flags to be protected by the designing team. This provisioned chip and associated protected firmware will be generated by running the build tools a second time. To protect the real provisioned chip, it would be wise to change secret parameters and values each time the build tools are run so that the "evaluation" files and the provisioned chips do not share secret values.

To request a provisioned chip for a specific team, email ectf@mitre.org with the subject line "*Provisioned Chip Request – <Team Name>*", where *"<Team Name>"* is the name of your team. The body of the email should specify which team's design you would like provisioned and a shipping address for the chip. Please allow up to two days for us to ship the chip, but we will attempt to process each request as quickly as possible.

At any point in time a team may only have two provisioned chips that they have yet to capture a flag from. Once a flag has been captured, a new provisioned chip from another team may be requested. Due to the limited number of un-attacked chips a team may have out at a time, chips should only be requested after a proof-of-concept attack has been developed.

We strongly encourage responsible disclosure[9] if any vulnerabilities are discovered on open-source or commercial components used as part of the system. If desired, MITRE can help to coordinate the responsible disclosure of weaknesses to appropriate parties.

7 Scoring

Points are primarily scored in one of the four ways listed below.

7.1 Retrieving and submitting flags to MITRE

Each system is required to hold and protect "flags" that should only be revealed if the system is compromised. By submitting flags, a team is demonstrating that they have compromised the target system. A brief description is required for each attack that results in a flag submission. The point value of any given flag will be adjusted dynamically and automatically based on multiple factors:

- If multiple teams capture the same flag, then that flag will be worth less points than if only a single team is able to capture it. Naturally, more difficult attacks will be executed by fewer teams and therefore rewarded with more points.
- The number of points for a flag increase as time goes on without anyone capturing it. This will make the difficult flags more and more appealing as the competition goes on.
- To prevent teams from "holding" onto a flag without submitting it, the team that captures each flag first will get significantly more points for that flag than teams that capture it later.

[9] https://en.wikipedia.org/wiki/Responsible_disclosure

The lowest point value for a flag (i.e. one extracted during the first day) is worth 500 points. An unclaimed flag on the final day of the competition is worth 1575 points. The first team to submit a flag to the scoreboard will gain a large number of "shares" for that flag. From that point forward, any other team to submit the same flag will receive fewer shares based on the amount of time that has passed, where longer periods of time after the initial capture mean fewer shares. A team will score points for a captured flag equal to:

```
    <value of flag> * (<team flag shares> / <total flag shares>)
```

A single team claiming a flag will score points equal to the full flag value. The last team to claim a flag that has been claimed 10 times in total will score at most points equal to 1/10 of the flag value, though fewer points are more likely.

7.2 Protecting flags from attacking teams

Teams will be awarded points each day for every flag that has not been captured by other teams. As a result, more secure designs are likely to accrue more points than other designs. Additionally, teams that submitted a design that does not meet the functional requirements during the Handoff Phase will have less opportunity to accrue points. An unclaimed flag on the final day of the competition will have accrued 1075 points over the course of the competition (assuming the design was accepted on the first possible day). A flag claimed on the first day would score zero (0) points for the defending team.

Points are primarily scored in one of the three ways listed below. Additional points may be awarded at the discretion of the eCTF team (e.g. exemplary documentation). Further details on scoring will provided closer to the Attack Stage.

7.3 Documentation

Good documentation will be rewarded to discourage security-by-obscurity. "Good documentation" is meant to describe clear and well-commented code, useful descriptions of modules/functions/classes, and other documents that clearly describe how to read or approach the entire code base. **We are not looking for justification of your design or lengthy documents detailing your implementation in excruciating detail.** ***A concise and clear README.md, combined with well-structured and well-commented code can be sufficient for Max points. Quality will be valued over quantity.***

Possible points for documentation are as follows:

- **Max (1575 pts):** Exemplary documentation, comments, and code structure that is clear and easy to understand. This max point value is equal to the number of points for capturing a flag captured on the last day of the competition (and with full shares in that flag).
- **75% (1181 pts):** Good comments and high-level documentation
- **50% (787 pts):** Good comments, but lack of clear high-level documentation
- **25% (394 pts):** Confusing code and little or no actual documentation
- **0% (0 pts):** Very confusing or deceptive comments and documentation

Points for documentation will not be awarded until near the end of the attack phase to allow for proper analysis. Honest feedback on documentation from other teams is welcome.

7.4 Write-ups

There will be an opportunity for the top five (5) teams to provide write-ups for additional points. These teams will have an opportunity to submit a defensive write-up as well as a single attack write-up. The defensive write-up may discuss security measures that worked well, those that could have been improved upon, or any that were planned but could be developed in the time provided. The attack write-up is to award teams that develop interesting or novel attacks which do not directly capture an existing flag. Further details on the number of teams that may submit write-ups and the content/format of the write-ups will be provided later.

7.5 Flag Descriptions

The format for any discovered flag will be of the form `{<flag_name>_<16 hex characters>}`. For example, '`{malicious_firmware_0123456789abcdef}`'. The specific flag names can be found below.

The following table lists the flags, as well as a description of each:

Name	Description	Requirement
Rollback	The initial firmware on all provisioned chips will be version number 2. A protected firmware image with version number 1 will be provided which has the Rollback flag in it. If you can get the version 1 firmware to install and run, the flag will be written to serial. (Version 1 shouldn't be possible to load and run since the initially provisioned firmware has version number 2) Flag name: `rollback`	Versioning
Invalid Firmware	Submit any firmware image that will load and start running that is different than any of the protected firmware images that are posted or provisioned on the chip. (Note: this proves there is a flaw in the firmware integrity checking system.) Submission of this flag will require coordination the eCTF team; see below for submission details.	Integrity
Malicious Firmware	Provisioned chips will have a flag in memory that is accessible by firmware. If you are able to install and run custom firmware (through any method), you could then dump the flag in memory. An example (unprotected) firmware image will be available that is capable of reading this memory. (Note: this proves the integrity checking either has a flaw or can be bypassed.) Flag name: `malicious_firmware`	Integrity
Intellectual Property	The provided, (protected) firmware image version number 1 will contain a flag embedded somewhere in its memory. It is possible to retrieve this flag by defeating the confidentiality requirement of the system. Flag name: `intellectual_property`	Confidentiality
Readback Sniffer	The readback tool must be designed to resist attacks from eavesdroppers that may capture interactions between your readback tool and your bootloader. A recording of a flag being read out of memory by the bootloader and sent to the readback tool will be provided with provisioned chips. If the communication is not properly protected, the flag will be revealed. Flag name: `readback_sniffer`	Readback
Memory Read	Pull out the initial firmware loaded on the provisioned chip; somewhere in the firmware will be the flag. This firmware (version 2) will never be posted on the update site so only dumping the provisioned firmware will give access to this flag (i.e. defeating the firmware protection won't help.) Flag name: `memory_read`	Readback (and set your locks bits and fuses!)

Unlike the other flags, the Invalid Firmware flag does not exist in the provisioned chip, nor is it in a provided firmware. To get this flag, a firmware image must be submitted to the eCTF admin team that will load onto the provisioned chip using that team's own "Firmware – Update Tool" (i.e. no other software or hardware intervention needed). To submit this firmware, email ectf@mitre.org with a subject line "*Invalid Firmware Submission – <Team Name>*", where *"<Team Name>"* is the name of your team. Upon receipt, MITRE will attempt to load the provided firmware (additional contact with the submitting team may be necessary). Once the firmware has been confirmed to load, the team will be awarded with the Invalid Firmware flag on the online scoreboard backdated to the time of the original submission email.

8 Award Ceremony

All teams (students and faculty advisors) are invited to an award ceremony at MITRE on April 20th. This event will be collocated in Bedford, Massachusetts and McClean, Virginia - your team may attend at whichever location is more convenient. During the award ceremony, the top 5 teams will be invited to give a presentation of their work during the design and attack phases of the competition. At this time, the final scoring will be revealed, including points for the write-ups, and awards will be presented.

9 Important Dates

Kickoff --- January 18th, 2017

- Competition officially kicks off.

System Hand-off --- March 1st, 2017

- System design and implementation is due.
- After MITRE has verified a submitted design, the designing team will be given access to all other verified designs for attack.
- Scoreboard opens.

Scoreboard Closes --- April 14th, 2017

- Flag submission is closed.
- Teams will be contacted for write-ups, which will be due April 18th

Award Ceremony – April 20th, 2017

- All teams will be invited to MITRE for an award ceremony, where MITRE will announce the results of write-up judging and present awards.

10 Rules

Most rules are described and explained throughout the challenge description in the earlier sections, but this section serves as a concise summary of the most important rules.

(1) In addition to the rules provided by MITRE, participants should also adhere to all the policies and procedures stipulated by their local organization/university.

(2) MITRE reserves the right to update, modify, or clarify the rules and requirements of the competition at any time, if deemed necessary by the eCTF admins.

(3) When submitting your secure design, all source code and documentation must be shared.

  (a) This is to discourage security-by-obscurity, as well as to accelerate attack development and encourage more sophisticated techniques for both sides.

(4) During the attack phase, only attack the student-designed systems explicitly designated as targets.

(5) All flags must be validated by submitting a brief description of the attack.

  (a) Attack descriptions should be sufficiently detailed to allow the defender to correct their vulnerability.

  (b) eCTF admins may invalidate points for flags that are not validated before the completion of the eCTF.

(6) No permanent lock-outs are allowed. No delays longer than 5 seconds per boot or update are allowed.

(7) Team sizes are unlimited.

  (a) Most teams will consist of members of varying degrees of experience and skill level. Our hope is that this creates an opportunity for mentoring, where the most knowledgeable team members will help teach and guide the other members of the team. Team advisors should help manage meeting times and organization of large teams.

  (b) We want to encourage as many students to participate as possible, even if they are not willing to commit a significant amount of time to the competition.

  (c) An unlimited team size is fairer for competition since enforcing team size is difficult/impossible.

(8) Teams may consist of students at any level: undergraduate, graduate, PhD, or a mix.

(9) Teams are limited to two write-up submissions.

If you have any questions, ask!

  (a) Join our slack channel: ectfmitre.slack.com

  (b) Email: ectf@mitre.org

11 Frequently Asked Questions

11.1 Is it OK to obfuscate our source code to make it more challenging to understand and attack?

No. Obfuscations performed at compile-time (e.g. to make binary reversing more challenging) is OK, but your source code needs to be written in a clear and maintainable fashion. It should be well commented and/or otherwise documented clearly.

11.2 Can we add intentional delays during boot or firmware updating to make it more difficult for an attacker to collect large numbers of observations?

There should not be any intentional delays that may be noticeable to the user because a slow boot or update time will negatively impact the user experience and hurt sales for your product. For our purposes, we'll consider any delays more than 100 milliseconds to be noticeable to the user.

If your system detects that it is under attack, additional delays are OK, but must be limited to no more than 5 seconds per boot or update. Permanent lock-outs or self-destruction is not allowed (see next question).

11.3 Is it OK to brick the board when an attack is detected?

No! This chip will be going into a car! We can't have it be so easy for an attacker to disable the entire car! Can you imagine the cost of the recalls?!

11.4 Can we physically modify the chip with countermeasures?

No. We provision the chips, so you won't have an opportunity to modify the chip during provisioning. Everything that needs to be done to the chip for provisioning needs to be done in an automated fashion by your Build and Configure tools.

11.5 How many chips can we get during the attack phase? (e.g. if we keep bricking them, can we keep getting new ones?)

Each team will be limited to having two provisioned chips at a time. Once you've successfully captured at least one flag, you may request additional chips, but MITRE may request that you return the chips that were already attacked.

11.6 Can we attack the other teams' development environment?

No! Everything other than the provisioned chips, the host tools, and the firmware images are considered out-of-bounds. In other words, there is **nothing** that you are allowed to attack until we get to the attack phase.

11.7 Is social engineering in-scope for this competition? Can we send phishing communications to other teams to trick them into revealing their secrets?

No, please don't do this. Keep your attacks technical. ☺ We love creative ideas, but this one can easily violate university, state, and federal regulations.

12 Extra Tips

Remember that the last 512 bytes of EEPROM are reserved for secret values used by the application firmware. Any data that is present in the reserved bytes in `eeprom.hex` will be overwritten.

Intel HEX files can be generated using the avr-objcopy command. The example Makefile does this using the following two commands:

```
avr-objcopy -R .eeprom -O ihex bootloader.elf flash.hex

avr-objcopy -j .eeprom -O ihex bootloader.elf eeprom.hex
```

The first command takes every section except for the EEPROM and puts it into `flash.hex`. The second takes the EEPROM section and writes it to `eeprom.hex`. Note that the bootloader is compiled into `bootloader.elf` using `avr-gcc`.

13 Changelog

1/18/2017 – Initial release

2/28/2017 – Update for additional information on scoring, hand-off, and attack phase

- **Section 4.4.1 - Bootloader – Build Tool,** added (optional) instructions to generate .hex files for AVRDUDE that are not specified under the API.
- **Section 5 - Handoff Phase**, updated to clarify submission instructions including the necessary structure of a submitted design.
- **Section 6 - Attack Phase,** updated for clarification and details on:
  - Available files for “evaluation”
  - How to access the scoreboard and opposing team “evaluation” packages
  - Requesting provisioned chips
- **Section 7 - Scoring**, updated for clarification and details on:
  - Specific point values and scoring algorithm
  - Scoring criteria and points for documentation
  - Accessing and identifying flags
- **Section 8 - Award Ceremony**, (new section)
- **Various –** Minor fixes for clarification, typos, etc.