Hack@CHES 2021

Phase 1

Document for Participants

Table of Contents

Introduction

Hack@CHES is a CTF competition for hardware RTL designs. There are two phases in the competition: 1) phase 1 - before CHES, and 2) phase 2 - at CHES. In the phase 1 of the Hack@CHES 2021 competition, teams from both academia and industry will be competing to find hardware security bugs in the provided SoC design. By June 7th, all teams will receive the SoC design and relevant documentation. Teams will then have until August 16th to find security bugs using any method available unless any that are specifically forbidden in the rules section of this document. Once found, teams can submit a bug according to the submission guidelines provided in this document. These bugs will be scored based on a number of factors such as security impact and detection difficulty. Once phase 1 is over, the final scores and the top performers of phase 1 will be announced. Phase 2 is organized virtually during the CHES 2021 conference.

General Information

Timeline

The Phase 1 of the competition will start on June 7th 2021 and the teams will have till August 16th at 11:59 am PDT to submit the bugs.

Rules

We expect that both advisors and team members ethically adhere to the rules specified below. Advisors are responsible for ensuring that their teams abide by these rules:

1. Only participants of each team can work together in phase 2.
2. Teams may not communicate with each other regarding this competition.
3. Teams are not allowed to compare the RTL and design of the provided SoC with similar open-source SoCs.
4. Be professional and patient when contacting organizers and other teams.
5. Organizers have the right to disqualify teams/participants at their discretion. Judges' decisions are final.
6. Do not share details about the bugs you find or how you found them with anyone that is not in your team, except for submitting the bugs through the google form. This is a competition, not a collaboration.
7. Be considerate of the judges. Each bug submission takes time to review, so do not submit bugs that clearly are not security bugs or multiple copies of the same bug.
8. Contact your team before you submit a bug. This will prevent a duplicate submission of a bug.

9. Do not share the google form, SoC, competition email, slack channel, or documentation with anyone outside of the competition.
10. Do not modify the provided toolchain.

Scoring

The scoring for the Phase 1 works as follows: teams will submit the bugs through the google form, which is time stamped on submission. Not all bugs are scored equally. The bugs are scored based on multiple factors determined by the judges, such as the difficulty to find a bug and its severity from a security standpoint. Note that **points will be awarded only for reported bugs with clear security impact not for functional bugs in design**. Security bugs are design/implementation issues that enable untrusted agents (as defined in the adversary model below) to bypass security features or compromise protected SoC assets. Bugs submitted that are unclear or unspecific will not be scored, so make sure you are very clear and unambiguous in your bug submission.

Unique Bugs

Note that you can score a maximum of **60** points with each unique bug submitted. The score breakdown is as follows:

- **5 pts** - For confirmed issue.
- **5 pts** - For the security impact severity assessment(CVSS), and test submitted (test is the output of simulation, test script or any other information confirming the issue).
- **50 pts** - For writing user level code that exploits the bug found.

Note:
1. For each unique bug found in ROM, bonus **20 pts** will be awarded .
2. For physical attacks, an attack flow description with simulation test script that demonstrates the feasibility of the attack is needed.
Please report all the above information to score maximum points for each reported bug.

Bug Variants

Non-unique bugs, called bug variants, are bugs that share a similar root cause to a bug that was already submitted but was reapplied to a new application. The score is as follows:

- **3 pts** - For a variant of a unique bug already submitted.

Exploit Variants

Teams can submit multiple exploits for the same bug and its variants. Points will only be given for each new exploit if they are considered significantly different (For example: exploit of same vulnerability through different attack points, chaining of multiple vulnerabilities).

- **15 pts** - For writing user level code that exploits the bug found. For physical attacks, an attack flow description with simulation test script that demonstrates the feasibility of the attack is needed.
- **0 pts -** For reuse of the same exploit on different bugs.

Automation

We would like to encourage teams to develop tools that can generate exploits automatically and the scoring reflects this. They can be existing tools or unique tools created in-house. To receive the bonus points for a tool, contestants must prove the use of the tool to generate the exploit. The bonus of using tools are as follows:

- **100 pts** - For the first time the team uses a custom automation tool or methodology that they have developed themselves to generate exploits.
- **50 pts** - For the first time teams use an existing automation tool or methodology to generate exploits.
- **5 pts** - For tools that can automatically detect vulnerabilities.

If you have any questions about if your tool meets our requirements for the bonus points please contact the organizers before developing or using the tool.

Scoring examples

Here are some scenarios to further explain how the organizers will score the bug submissions.

- Scenario 1:
    - Manually written test bench for vulnerability identified in RTL.
    - Run test in RTL simulator such as ModelSim, Verilator, etc.
    - Submission includes: valid finding, correct description of root cause, simulator snapshot as tool output, correct root cause, severity evaluation.
    - Write exploit code to demonstrate vulnerability
    - The bug found is not in ROM
    - Total points = 5 for valid issue + 5 for corect root cause and security impact assessment + 50 for exploit code => Total 60 points.
- Scenario 2:
    - Same as scenario 1, except the bug found is in ROM.
    - Total points = 5 for valid issue + 5 for corect root cause and security impact assessment + 50 for exploit code + 20 bonus => Total 80 points.
- Scenario 3:
    - Manually written test bench for vulnerability identified in RTL.
    - Run test in RTL simulator such as ModelSim, Verilator, etc.
    - Submission includes: valid finding, correct description of root cause, simulator snapshot as tool output, correct root cause, severity evaluation, valid mitigation idea.

    - No exploit code provided.
    - The bug found is in ROM
    - Total points = 5 for valid issue + 5 for corect root cause and security impact assessment + 20 bonus => Total 30 points.
- Scenario 4:
    - Variant of exploit code for a previously submitted bug where this exploit uses a different attack point and is not a simple modification of previously submitted exploit.
    - Total points = 15 for exploit code => Total 15 points.

In case of a tie, the time of bug submission will be used as a tie breaker. **ALL DECISIONS ON BUGS AND FINALISTS IN THE COMPETITION ARE AT THE DISCRETION OF THE JUDGES.**

Adversary Model

#	Type	Description
1	Unprivileged software at user-level mode	Executes on core with user-level privileges but may exploit bugs to mount privilege escalation attacks
2	Physical attacker	Has physical possession of the device
3	Privileged software at supervisor mode	Executes on core with Supervisor mode privilege but may target other higher privilege levels
4	Authorized debug access	Has the ability to unlock and debug production device

Bug submission procedure

Teams should use the provided google form to submit individual bugs. In the event of a tie, the time of submission will be used to determine the winner. Please fill in all fields in the submission form. If multiple bugs are inherently linked but have different target/effect, report them separately. Do not submit multiple bugs together. Please follow the instructions provided at each field in the submission link.

Submission link

https://forms.gle/SVaxCj4kCdwdZRiL7

Example submission

Team name	Bug stompers
Bug number	1
Security feature bypassed	JTAG password protection
Finding	Signal “pass_chk” indicates that password matches and is used to unlock the JTAG password checker. This signal is not reset to zero when the JTAG module is reset.
Location or code reference	./src/riscv-dbg/dmi_jtag.sv; lines 239-245
Detection method	Manual inspection
Security impact	Once the user provides the password, the JTAG will remain unlocked, even when the module is reset. The attacker can use this to read secure data from the chip.
Adversary profile	Physical attacker
Proposed mitigation	Add “pass_chk” to reset logic.
CVSSv3.1 Base score and severity	Medium (6.3)
CVSSv3.1 details	CVSS:3.1/AV:P/AC:H/PR:N/UI:R/S:U/C:H/I:H/A:H Access vector: Physical. Since physical access is needed for JTAG Attack complexity: High. The attacker has to find a debug unlocked system. Privilege Required: None. The attacker doesn’t require privileges. User Interaction: Required. A valid debug user unlock system once. Scope: Unchanged. Scope is limited to the system that is compromised by this vulnerability. Confidentiality, Integrity & Availability impact: High. In Debug mode all system assets are compromised by the attacker.

Competition files

All the documents and additional resources for participants can be found in the Hack@CHES'21 Google Drive directory[1].

The setup for the Phase 1 can be done in two ways:

1) Mount the VM image we provide which comes with the toolchain and verilator installed. You can start with simulations directly this way.
2) Following our instructions pdf to run a setup environment for Verilator simulations directly in your computer.

The detailed setup instructions are provided in the 'Setup' section of this document

[1] https://drive.google.com/drive/folders/1M8pNU5efFk2KhwX4svoPg4SLiM-bm9Ed?usp=sharing

Contact information

All teams will be added to multiple public slack channels and a private in the Hack@CHES slack workspace. The public slack channels are for general questions with respect to logistics (e.g. trouble setting the SoC up, clarifications on rules, etc.) that anyone can reply to and answer. It is not for discussions about bugs or methods to find them. All participants are suggested to use and interact in the slack channel. The private slack channel is for any specific/private questions. However, we will not be responding to questions related to how bugs should be detected or whether a certain scenario is a bug.

ALL ANNOUNCEMENTS DURING THE PHASE 1 OF THE COMPETITION WILL BE MADE PRIMARILY THROUGH SLACK. Hence, we strongly suggest that you join the slack workspace.

Slack channel

Name**:** hackches2021.slack.com

Invite link:

https://join.slack.com/t/hackches21/shared_invite/zt-rbwa34aq-HcQJLythkFMTI3Bm2avl4g

Email

hackatevent@gmail.com (When sending messages, please specify which competition you attend.)

Website

https://hackatevent.org/hackches21/

Information about the SoC

The competition uses Openpiton, an open source, general purpose, multithreaded manycore processor. Ariane processors are used as the core. We use a 2 core configuration for the competition (x_tiles = 2, y_tiles = 1). Core with ID 0 is the primary core the executable runs whereas core 1 is used as DMA (more in DMA section below).

Peripherals of the SoC:

- Bootrom
- Debug module
- UART

- JTAG
- SD
- DRAM Ctrl
- Ethernet
- CLINT (Core-local Interrupt Controller)
  - For timing and local interrupts
- PLIC
  - Interrupt controller
- FUSE
  - FUSE memory that stores keys/passwords and all other configuration information.
- AES0
  - AES0 is implemented in CTR mode.
- AES1
  - AES1 is a symmetric block cipher AES
- AES2
  - AES 2 is a Galois/Counter Mode (GCM) AES.
- SHA256
- DMA
- HMAC-SHA256
- PRNG
- RSA
- Reset Controller

Simulation

Setup and simulation is for the bare-metal SoC where the application runs in user-level privilege. Use the setup guide provided in the Hack@CHES'21 Phase 1 stage google drive for setup instructions.

Security Features

The SoC is augmented with the following security features:

- Proxy kernel:
  The proxy kernel is a lightweight execution environment that handles I/O related system calls by proxying them to the host computer[2]. It enables virtual memory addresses and memory isolation. Proxy kernel includes firmware code that runs in the Machine mode that loads the peripherals with the data in the FUSE memory.

  The executable of the user program is embedded into the proxy kernel elf to generate a single executable that begins with running pk, switches to user privilege level

[2] https://github.com/riscv/riscv-pk/

and runs the user application. Refer to the setup instructions for instructions related to generating the executable with both pk and user program embedded in it.
PK can be modified for debugging purposes, but the final bug/exploit submitted should be valid with the original pk since the judges evaluate the submissions only on the original pk.

- AES0 engine:
  AES0 runs in Counter-mode taking a 192-bit key, 128-bit input data (plaintext/cipher text), and 128-bit initial vector and produces 128-bit output (ciphertext/plaintext respectively). There are three AES keys provisioned, where the desired key can be selected depending on how the AES engine is called, i.e., arguments to the system call used.

- AES1 engine:
  This AES engine supports 128 and 256 bit keys. The implementation is iterative and processes one 128 block at a time. This AES engine also has three AES keys provisioned from which one has to be selected through arguments to the system call.

- AES2 engine:
  AES2 runs in the GCM mode of operation for symmetric-key cryptographic block ciphers. It uses one of three 128-bit keys, a 128 bit input plaintext input, and a 128 bit initial vector to output a 128 bit cipher text.

- SHA-256 engine:
  The SHA-256 engine takes a 512-bit input and produces a 256-bit hash value. It can handle larger input messages (data length > 512 bits), by splitting them into chunks of 512 bits. The SHA-256 engine can be called by means of a system call.

- HMAC-SHA256 engine:
  HMAC is a message authentication code (MAC) that uses both a hash and a secret crypto key. Our HMAC engine uses the SHA-256 module for the calculation of the HMAC. This engine is added as a peripheral and takes a 512-bit input, 256-bit key, and produces a 256-bit hash value. The HMAC engine can be called using a system call.
  The message has to be exactly 512 bit input. It can use the key directly or use the key hashes precomputed to skip the first rounds of sha-256.

- Secure debug:
  JTAG is protected with a password checker which prevents the use of all JTAG operations unless the correct password is provided by the user. This password is sent to the HMAC module and hashed. This hash is then compared to the hash of the correct password and is unlocked if they match. The passwords/keys of all the peripherals are reset in debug state to avoid leaking.

- FUSE memory:
  Some of the SoC peripherals require a key/password or configuration values to be provisioned for their operation. The set of private keys/passwords or configurations are available in the FUSE memory. During the firmware setup phase, the data in the FUSE memory is transferred into the respective peripheral registers with the help of the address look-up table present in the PKT peripheral. This address look-up table in the pkt peripheral contains the address of the peripheral register corresponding to the data stored in the FUSE memory.

- Access control in the system bus:
  The cores have different access permissions to the various peripherals in the SoC. These permissions are different for different privilege levels of the primary processor (core0) and cannot be modified.

- DMA engine:
  The DMA engine allows peripherals to access the main memory subsystem to perform direct memory accesses without involving the primary processor. The implementation is done using a set of registers connected as a DMA peripheral where control signals are communicated by the primary processor (core0) to the secondary processor (core1). The job of the secondary processor is only to perform DMA transfers.

- RISCV PMP protections:
  The memory accesses through DMA engine are protected by a mechanism similar to the PMP register protection from RISC-V specification. This mechanism uses the values from the pmp CSR registers. The design does not implement the actual PMP protections on MMU but just on the accesses made by the DMA engine. Refer to Ch. 3.6.1 from the RISCV Privileged specification.

- Peripheral locks/Register locks:
  To prevent malicious use of resources, registers of some peripherals are locked behind register locks. These locks prevent complete access to the peripherals irrespective of the processor privilege level. The registers that can potentially leak secret data are locked by register locks. The following peripherals can be locked by register locks: AES0, AES1, SHA-256, HMAC, PKT, Access Control, Register lock and DMA

- Crypto test on bootup:
  During bootup, the crypto engines are tested to ensure that they are functionally correct and have not been tampered with in the Bootrom. If they are found to be incorrect, the processor stops further execution by entering an infinite loop in the Bootrom.

- PRNG:
  The SoC consists of a Pseudo-Random Number Generator (PRNG) that is used by the users to generate random numbers through system calls. The PRNG has four entropy sources ranging from 16bits to 128bits. The 64bits output of PRNG will be combined by segments of four encrypted sources. In the user mode, people can only get a 64bits random number from PRNG.
- RSA:
  RSA is an asymmetric key cryptography that accepts two maximal 1024bits prime numbers to generate 2048bits encryption key, description key and modulus. The RSA module has two modes. One is encrypting original binary messages. The other one is decrypting ciphertext to get the binary messages. The default size of keys and modulus 64bits because the simulation speed of verilator is very slow (> 1hr). If we use modelsim, there is no such simulation speed issue. We can modify the size at the wrapper of RSA.
- Reset Controller:
  The reset controller is a module that can send reset signals to other peripherals. It takes in an ID as input and sends a reset signal to the corresponding peripheral. The IDs can be found in the example applications.

Note :

Various APIs are being provided for the phase 1 to help the participants better understand and use the security features. These APIs can be found in the ariane/software/ariane_api.c file. Example applications (using_api*.c) that explains the usage of these APIs is also included in the same directory.

Happy debugging!!!

Verilator Simulation Setup

SoC simulation setup is done in three stages:

- Stage 1: Install dependencies
- Stage 2: Download the required files
- Stage 3: Install files

Note: There are two different SoCs provided for the competition: openpiton SoC. Please refer to the "Additional Information" section for details about these SoCs.

Stage 1: Install dependencies

For ubuntu systems:

```
$ sudo apt-get install autoconf automake autotools-dev curl
libmpc-dev libmpfr-dev libgmp-dev libusb-1.0-0-dev gawk
build-essential bison flex texinfo gperf libtool patchutils bc
zlib1g-dev  device-tree-compiler pkg-config libexpat-dev
```

For Fedora-based systems:

```
$ sudo dnf install autoconf automake @development-tools curl dtc
libmpc-devel mpfr-devel gmp-devel libusb-devel gawk gcc-c++ bison
flex texinfo gperf libtool patchutils bc zlib-devel expat-devel
```

Stage 2: Download the required files

Download openpiton.zip from the google drive

```
$ mkdir hackches21
$ mv openpiton.zip hackches21/
$ cd hackches21
$ unzip openpiton.zip
```

Stage 3: Install files

- Inside the hackches21_phase1 folder:

```
$ mkdir tools
$ mkdir tools/openpiton_tools tools/riscv_gcc
$ cd tools
$ export TOOL_DIR=$PWD
$ git clone --recursive
https://github.com/riscv/riscv-gnu-toolchain.git
$ cd riscv-gnu-toolchain
$ ./configure --prefix=$TOOL_DIR/riscv_gcc
$ make
$ cd $TOOL_DIR/riscv_gcc/bin
```

```
$ pwd
```

- Use the path from the output of the above command and replace it with the dummy path for the PATH variable in line 11 of hackches21/openpiton/pk/make_embedded.bash
- In hackches21/openpiton/piton/ariane_setup.sh, at line 79, change the path of the RISCV variable to path of the tools/openpiton_tools directory created above
- In hackches21/openpiton/piton/ariane_build_tools.sh, at line 91, change the path of the prefix from $ROOT/tmp/riscv-tests/build to <path to "tools" directory created above>/riscv-tests/build
- `$ cd hackches21/openpiton`
- `$ source piton/ariane_setup.sh`
- `$ ./piton/ariane_build_tools.sh`
- `$ cp make_run_user_with_pk.sh build/`

Running openpiton SoC

There is a script make_run_user_with_pk.sh that generates the user program elf file and then merges it with the proxy kernel to create a final executable. Then it is copied into openpiton/build directory and the simulation is run. The user program C files should be placed in openpiton/software directory.

- Every time you create a new terminal you need to run the following commands inside the hackches21 folder:
  ```
  $ cd openpiton
  $ source piton/ariane_setup.sh
  ```
- To run simulation:
  ```
  $ cd $PITON_ROOT/build
  $ sims -sys=manycore -x_tiles=2 -y_tiles=1 -vlt_build -ariane
  $ source make_run_user_with_pk.sh <name of the c user program>
  ```
  Ex: `$ source make_run_user_with_pk.sh helloworld`

- Note that the verilator simulations are slow. Depending on your machine, the simulation can take upwards of 10min. You can do $ tail -f fake_uart.log to keep track of progress through printf output.
- If you face any error, try running the following inside $PITON_ROOT/build directory:
  ```
  $ sims clean
  ```

- The printf output can be seen in fake_uart.log file. The trace files for the two cores are trace_hart_0.dasm and trace_hart_1.dasm

Note: You might have to hit Ctrl+C after the user program is done running to stop the simulation.

VM simulation Setup

The Virtual Image consists of everything required to start simulations which include the following:

- RTL simulation tool (Verilator)
- RISC-V Toolchain
- Buggy SOC (RTL)

Mounting the Virtual Disk Image

- Download and Install Virtualization Environment, like VirtualBox
- Download the OVA virtual disk from the Google Drive HackCHES-2021_VirtualMachine.ova
    - File -> Import Appliance (Select the downloaded OVA file)
    - Set the parameters for the Virtual Machine based according to your host system and click Import
    - Start the Virtual Machine.
- (If you are prompted for the UserID and Password use the following - UserID : hacevent Password : h@cKevent)

Running openpiton SoC

The script make_run_user_with_pk.sh generates the user program elf file and then merges it with the proxy kernel to create a final executable. Then it is copied into openpiton/build directory and the simulation is run. The user program C files should be placed in openpiton/software directory.

- Every time you create a new terminal you need to run the following commands:
  ```
  $ cd ~/Documents/hackches21/openpiton
  $ source piton/ariane_setup.sh
  ```
- To run simulation:
  ```
  $ cd $PITON_ROOT/build
  $ sims -sys=manycore -x_tiles=2 -y_tiles=1 -vlt_build -ariane
  $ source make_run_user_with_pk.sh <name of the c user program>
  ```
  Ex: `$ source make_run_user_with_pk.sh helloworld`

- Note that the verilator simulations are slow. Depending on your machine, the simulation can take upwards of 10min. You can do $ tail -f fake_uart.log to keep track of progress through printf output.

- If you have an error try running the following inside $PITON_ROOT/build directory:
  `$ sims clean`
- The printf output can be seen in fake_uart.log file. The trace files for the two cores are trace_hart_0.dasm and trace_hart_1.dasm

Note: You might have to hit Ctrl+C after the user program is done running to stop the simulation.