Possibility of arbitrary code execution by Step-Oriented Programming

Hiroaki Sakai

About Me

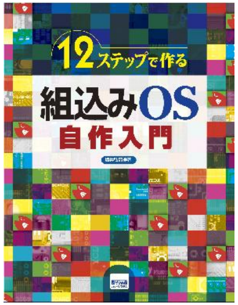
FUJITSU LIMITED NETWORK SERVICES BUSINESS UNIT
Fujitsu Security Meister (High Master Area)
Security & Programming Camp (Current Security Camp) Instructor
SECCON Executive Committee
SecHack365 Executive Council Committee
Program Committee for Workshop on Critical Software System
Personal Event Exhibition and many seminar talk
One of assembly language programming Tanka Rokkasen (six major poets)
Binary karuta creator
Free software creator
Professional Engineer (Information Engineering)

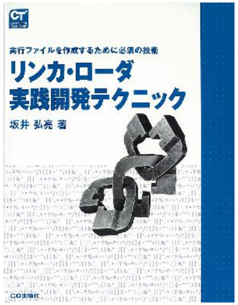
http://kozos.jp/

About Me

I made my own embedded OS "KOZOS" operating on an inexpensive microcomputer board as a hobby programming, implemented boot loader, simple multitasking kernel, device driver, simple TCP/IP stack, simple web server, debugger support and simulator support, and operated a web server with software written completely from scratch.

I also released them as open source software, exhibited and performed presentations at various events such as Open Source Conference.





ハロー
"Hello, World"
たった7行のCプログラムから解き明かす
OSと標準ライブラリのシゴトとしくみ
坂井弘亮[著]
あなたが最初に書いたプログラムにはまだ「知らないこと」がたくさんある。
printf()関数の先には何があるんだろう?
main()関数が呼び出される前には何があるんだろう?
main()関数からリターンする先はどこだ?
秀和システム

About Me

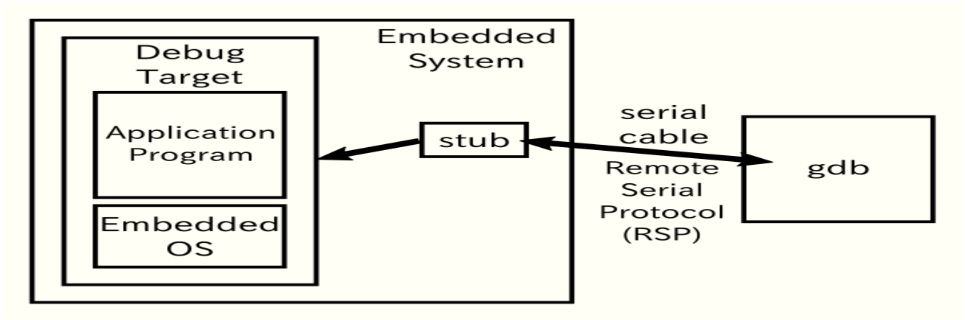
I am an embedded software engineer.

Today's topic is about what happens when an embedded software engineer tries security.

Abstract

Possibility of arbitrary code execution by Step-Oriented Programming

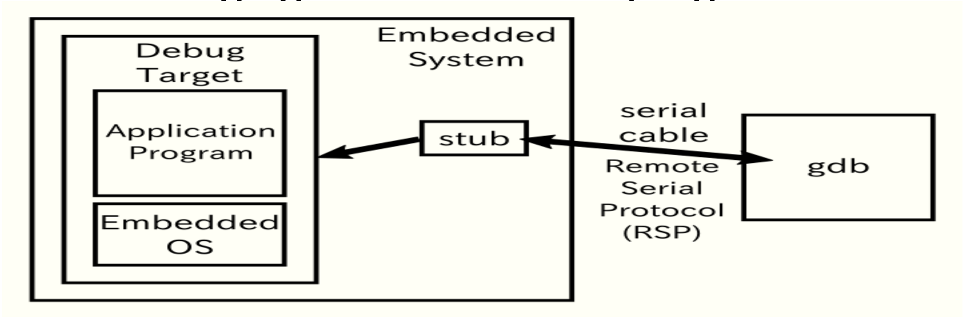
An embedded system has a stub to connect with a host PC and debug a program on the system remotely. A stub is an independent control program that controls a main program to enable debugging by a debugger. A stub is simplified by only processing the simple controls such as reading or writing of the register or of a memory, and a debugger processes a complicated analysis on the host PC.



Abstract

Possibility of arbitrary code execution by Step-Oriented Programming

Communication with a debugger on the host PC and a stub on the embedded system is performed by a protocol called Remote Serial Protocol (RSP) over a serial communication or TCP/IP communication. If this communication is taken away, it becomes possible to operate a stub arbitrarily. We considered what kind of attack possibility there was in that case, and identified that execution of arbitrary code constructed from pieces of machine code, combined with (SOP: Step-Oriented Programming) is possible by repeating step execution while changing the value of the program counter.

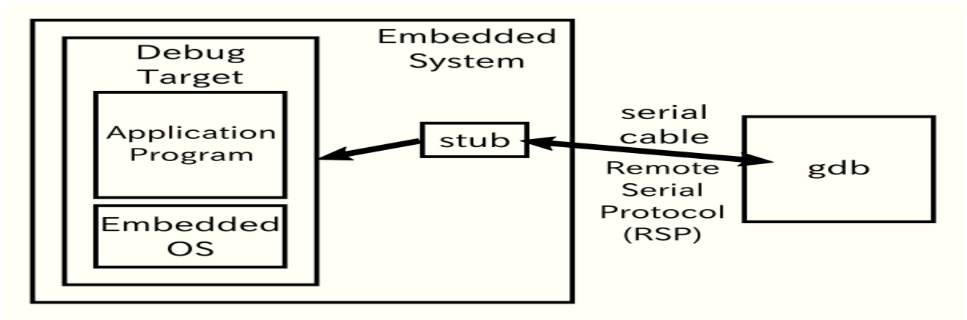


Abstract

Possibility of arbitrary code execution by Step-Oriented Programming

Therefore it is possible to construct an arbitrary code and execute it from existing machine code, even if execution of the injected machine code is impossible because execution on data area is prevented by DEP or only machine code on the flash ROM are allowed execution.

I will explain about an attack principle by SOP and the results from constructed attack code and actual inspection.



Agenda for Today

- What are embedded devices
- How embedded devices handle remote debugging
- Communication protocols for remote debugging
- Possibility of attacks by Step-Oriented Programming (SOP)
- Experiments at SECCON CTF

Important!

The purpose of this research is to be able to consider some essential defenses by understanding the possibilities of attacks when the connection was stolen, not to help any illegal attacks.

And, the purpose of this research is to enlighten security of embedded system by understanding danger.

It's illegal to attack or inspect others' server or equipment without permission. Don't do that!

Conclusion

- When the debugging port is open, anything can be done (of course)

- You cannot say that it is safe just because it is not possible to inject machine code
(Using SOP, it is possible to execute arbitrary code)

- Let's make sure that the debugging port is closed!
(If possible, try deleting stub code and removing the circuit itself)

What are embedded devices

What are embedded devices

- Rice cookers, air-conditioners, cars, bun steamers in convenience stores, ...
- Devices that are controlled at the software level by microcomputer

What are embedded devices

PCs and servers are "general-purpose devices"

- Users use them for generic purposes by installing applications (main factor is the applications)
- Resources can be expanded (users can replace CPUs or add more memories)
- Better to have lots of resources (too big is better than too small)

Embedded devices are "dedicated devices"

- Used for a specific purpose with specially built software (main factor is the devices themselves)
- Resources are limited (Users don't replace CPUs or add memories)
- Just the right amount of resources are required (right resources for the right tasks)

What are embedded devices (from a security stand point)

Controlled at the software level by microcomputers for embedded systems

- They are computer systems after all, like PCs and servers
- It is possible to get attacked same way as those

Operating everywhere (such as appliances, communication devices, smart meters...), where we don't even recognize

Compared to PCs and servers...

- Embedded devices handle very few information resources
- The number of embedded devices are huge
- Those are increased in recent years, but it isn't managed so much

Developing programs for embedded devices

Big differences between development environment and running environment

- Hosts and targets are different
  - Hosts may be modern PCs or servers
  - Targets may not be x86 or may be 8-bit microcontrollers

- Cross-Compilation
  - Complied at the host, executed at the target

- Remote debugging
  - Start a debugger at the host, connect to a debugging port of the target via ways such as serial cables and then debug

Debugging ports for embedded devices

Many embedded devices have a port for debugging

- That may be a serial port, may be a TCP/IP connection

Sometimes products have it secretly, or a circuit could be remaining which can be used by soldering a connector onto it...

- It is possible for embedded devices kept outdoors to be cracked during nights

Anything can be done if the debugging port is compromised (of course)

- Just like freely controlling programs operating in the debugger
- Rewrite memories, modify the program counter, etc
- It is possible to do anything such as injecting and executing shell code

So anything can be done when a debugging port is compromised

But what kind of attacks will actually happen...?

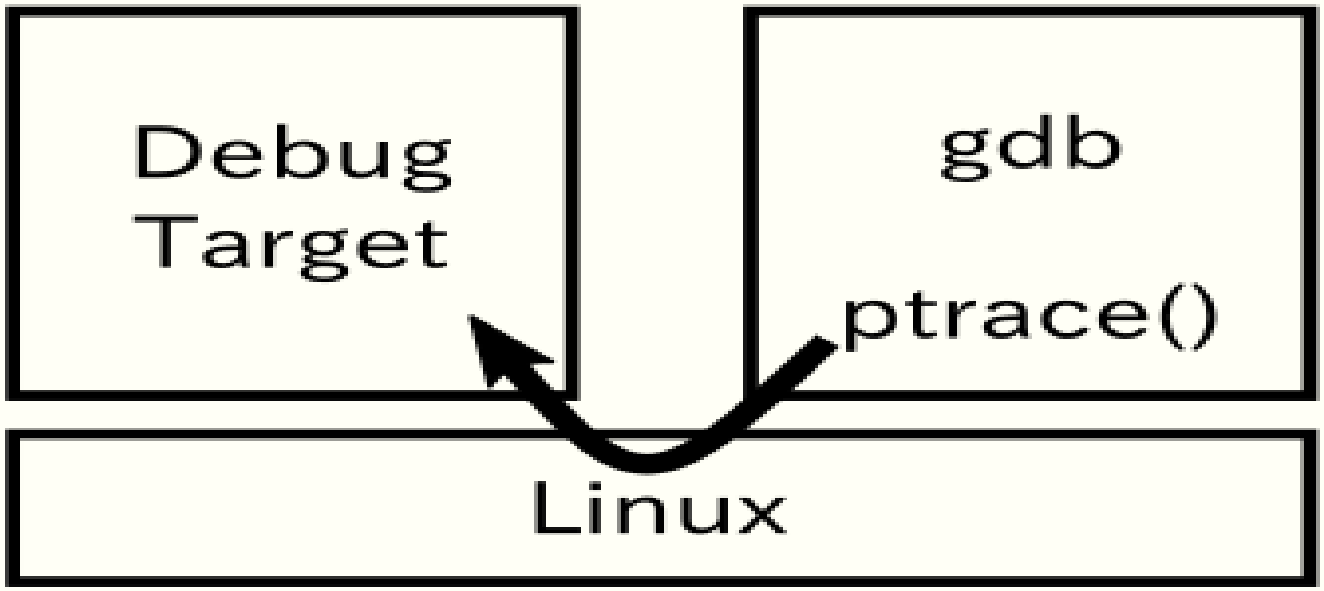
- Consider, in microcomputers there are architectures where it's essentially not possible to do things like injecting and executing code

Today's topic is to think about that possibility

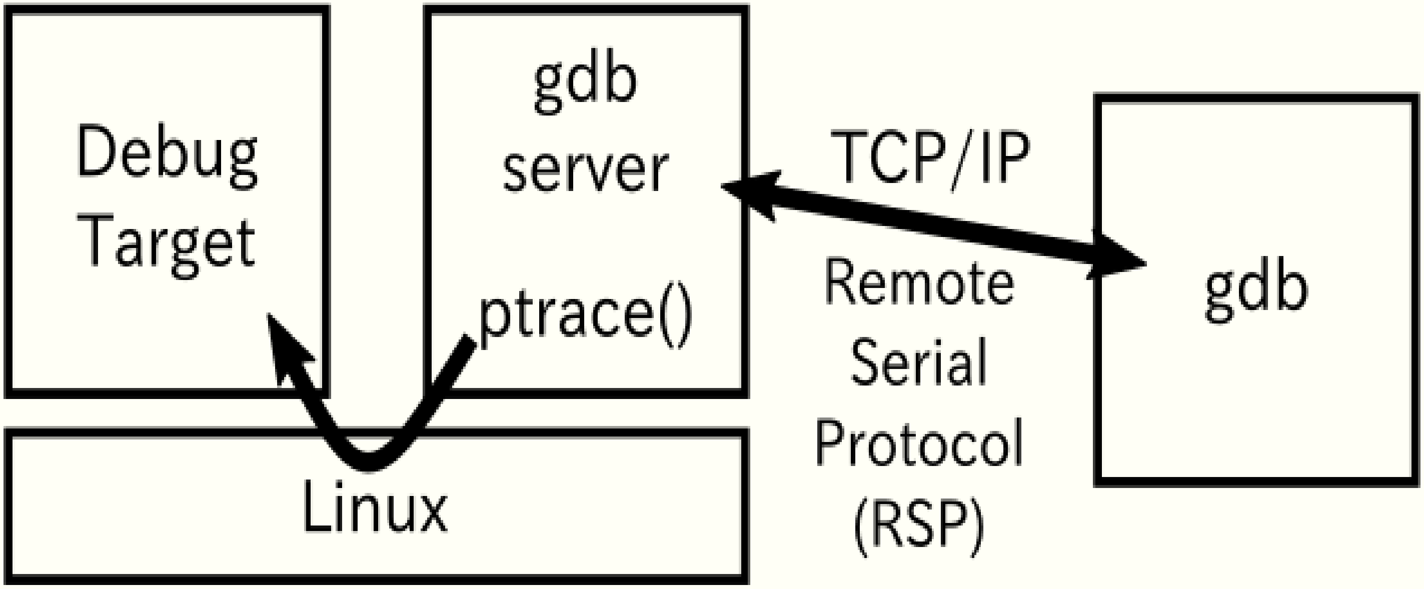
- I will not talk about how the debugging port is stolen
- The theme is to talk about what will be possible when the debugging port is stolen

How embedded devices handle remote debugging

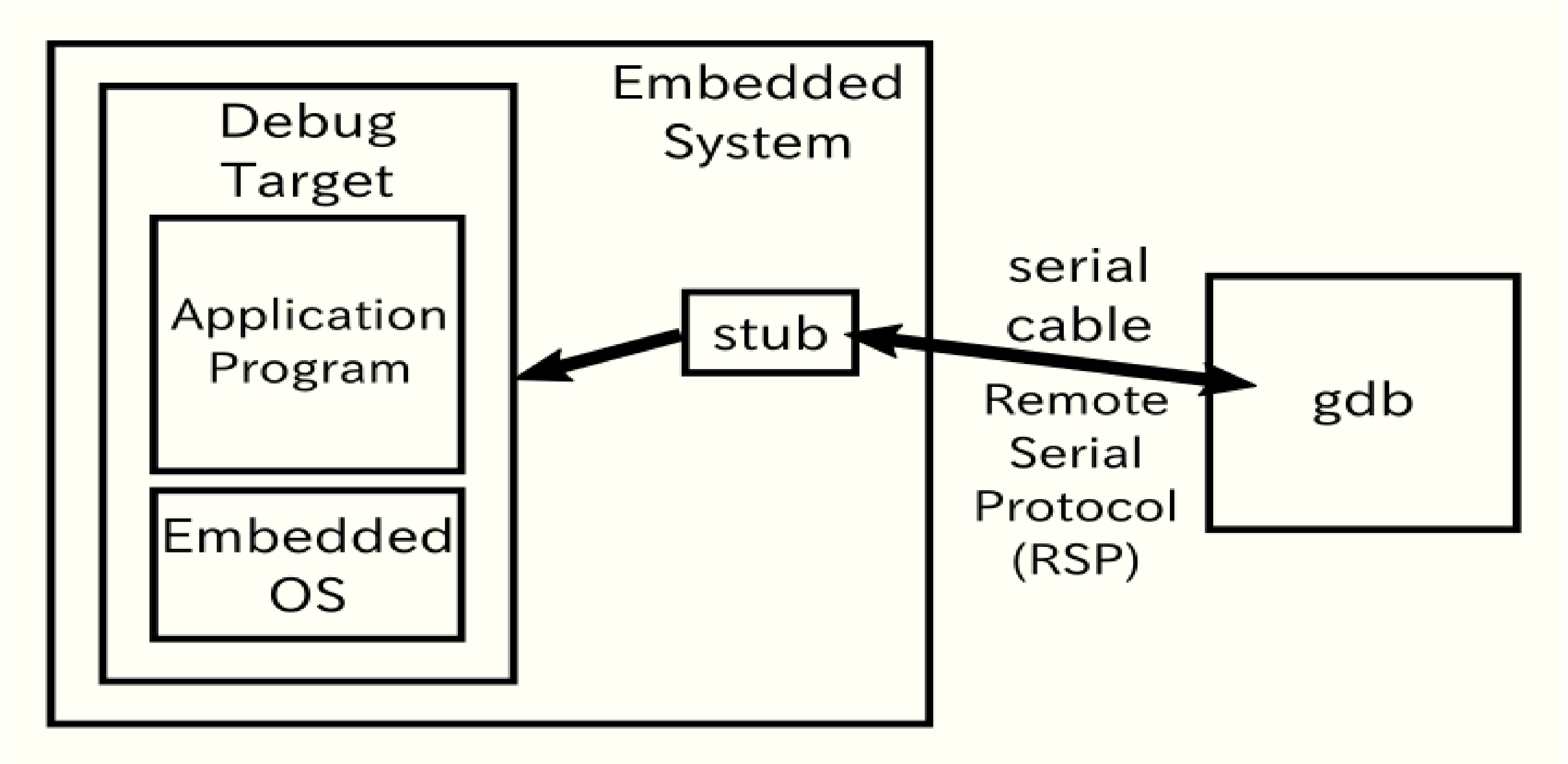
Normal debugging



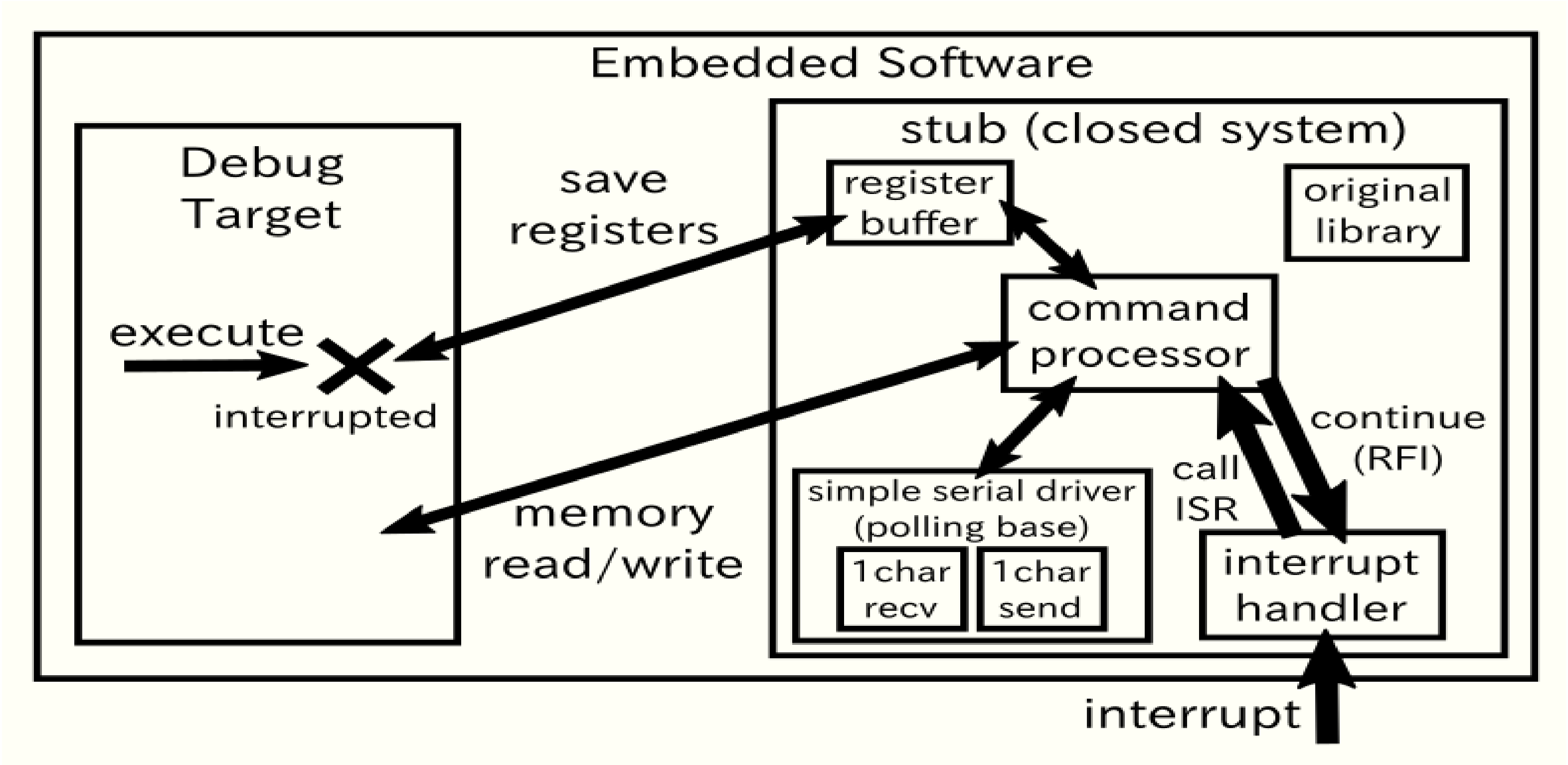
Debugging using gdbserver



Debugging of embedded devices



Stub structure



Stub implementation

```
p = recvbuf;
switch (*(p++)) {
...
case 'g':
  read_memory(registers, sendbuf, REGISTERS_SIZE);
  break;

case 'G':
  write_memory(p, registers, REGISTERS_SIZE);
  stub_strcpy(sendbuf, "OK");
  break;
...
```

Stub implementation

```
case ’m’:
  a2val(&p, &addr);
  if (*(p++) != ’,’) {
    stub_strcpy(sendbuf, ”E01”);
    break;
  }
  a2val(&p, &size);
  read_memory((void *)addr,
              sendbuf, size);
  break;
...
```

```
case ’M’:
  a2val(&p, &addr);
  if (*(p++) != ’,’) {
    stub_strcpy(sendbuf, ”E01”);
    break;
  }
  a2val(&p, &size);
  if (*(p++) != ’:’) {
    stub_strcpy(sendbuf, ”E02”);
    break;
  }
  write_memory(p, (void *)addr, size);
  stub_strcpy(sendbuf, ”OK”);
  break;
...
```

Communication protocols for remote debugging

RSP (Remote Serial Protocol)

This is a protocol to send and receive debugging information between a debugger on the host and stubs on the target

Only simple operation commands are defined

- Such as reading/writing of registers or of the memory
- The debugger on the host will handle complicated operations by combining commands

What you can do with RSP

- Get register values
- Set register values
- Read from the memory
- Write into the memory
- etc...

In short, you can do anything

You can inject executable code in the memory and execute them by modifying the program counter

Protocol for RSP

Format

```
$<command>{<parameter>}#<checksum>
```

Commands and parameters

- For example, "s" for the step execution, "c" to continue the operation
- Some commands have parameters after them (such as reading/writing of the memory)

Checksum

- The sum of the command + parameter part (1 byte)

Protocol for RSP

Response

- Return "+" as Ack when properly received
  (in a case like checksum error "-" will be returned as Nak and asks for a resend)

- Return "OK" when the command is executed

- Return "Enn" when an error occurs (nn is a number that describes the error type, Ex: E01)

- Return an empty command ("$#00") when the command is not supported

Example of RSP commands

Operations	Command columns
Step execution	$s#73
Get register values	$g#67
Read from the memory	$m2400,10#c0

Primitive commands

Commands	Meanings
`s`	Step execution (step)
`c`	Continue the operation (continue)
`g`	Get register values
`G`	Set register values
`m<address>,<size>`	Read from the memory
`M<address>,<size>`	Write into the memory
`D`	Detach

There are many other commands defined, but most debugging operations with gdb can be done using just the above commands

There are higher-level commands as well, but when they are not supported by the stub, gdb will automatically adjust to use other simple commands instead of using them

Demo
(Controlling the target using RSP)

Notes

- "Continue" in gdb does not only send c command to the stub
- In other words, gdb commands do not simply correspond to RSP commands

gdb combines various RSP commands
to materialize operations such as "continue"

Example of the "continue" operation

It is not just sending "c command" when continuing an operation from a breakpoint on gdb

To "continue" as expected, it needs to do the following operations (in the case of software breakpoints)

- With s command, do a step execution for just 1 instruction
- With m command, read and save the instruction which the breakpoint is placed on
- With M command, set the trap instruction to where we used break (if you didn't do this, you cannot properly break the next time)
- Use c command to "continue" operation

The "continue" works thanks to gdb that automatically executes these commands

Commands executed when setting a breakpoint with ARM

```
(gdb) set debug remote 1
(gdb) target remote localhost:10000
...
(gdb) break main
Sending packet: $m20c8,4#ca...Ack
Packet received: 04e02de5
Breakpoint 1 at 0x20c8: file arm-elf.c, line 39.
(gdb) continue
Continuing.
Sending packet: $Z0,20c8,4#13...Ack
Packet received:
Packet Z0 (software-breakpoint) is NOT supported
...
```

Test if Z0 command is available
(software breakpoint setting that can pass parameters)

-> Not implemented

Commands executed when setting a breakpoint with ARM

With m command, read machine code set at the breakpoint -> Save it

```
...
Sending packet: $m20c8,4#ca...Ack
Packet received: 04e02de5
Sending packet: $X20c8,0:#eb...Ack
Packet received:
binary downloading NOT supported by target
Sending packet: $M20c8,4:fedeffe7#e0...Ack
Packet received: OK
Sending packet: $vCont?#49...Ack
Packet received:
Packet vCont (verbose-resume) is NOT supported
Sending packet: $Hc0#db...Ack
Packet received:
Sending packet: $c#63...Ack
```

Test if X command is available (for fast reading/writing of the memory with binary communication) -> Not implemented

With M command, set the trap instruction at the breakpoint

Test if vCont command is available (Multi-thread extension for the "continue" operation) -> Not implemented

Test if Hc0 command is available (Specify the "continue" operation for each thread) -> Not implemented

Use c command to do the "continue" operation

Commands are progressively converted to simple ones

Commands executed when "continue" is executed with ARM (When a break occurs)

With g command, get a list of registers (It includes the program counter, so you can find where the breakpoint is)

```
Packet received: T05
Sending packet: $g#67...Ack
Packet received: 000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000001028000008200000c82000
000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000d3000000
Sending packet: $m20c8,4#ca...Ack
Packet received: fedeffe7
Sending packet: $qL1160000000000000000#55...Ack
Packet received:
Sending packet: $M20c8,4:04e02de5#0d...Ack
Packet received: OK

Breakpoint 1, main () at arm-elf.c:39
39      arm-elf.c: No such file or directory.
(gdb)
```

With m command, check the machine code at where the breakpoint is placed

With M command, recover the machine code at the breakpoint to their originals

Commands executed when "continue" is executed with ARM (When "continue" is executed)

```
(gdb) continue
Continuing.
Sending packet: $Hc0#db...Ack
Packet received:
Sending packet: $s#73...Ack
Packet received: T05
Sending packet: $g#67...Ack
Packet received: 000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000c28000008200000cc2000
000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000d3000000
Sending packet: $m20cc,4#f5...Ack
Packet received: 0100a0e3
Sending packet: $m20c8,4#ca...Ack
Packet received: 04e02de5
Sending packet: $M20c8,4:fedeffe7#e0...Ack
Packet received: OK
Sending packet: $Hc0#db...Ack
Packet received:
Sending packet: $c#63...Ack
```

With s command, do a step execution for one instruction (Stop at the instruction next to the breakpoint)

With g command, check the stopping position

With m command, get and save the instruction at which the breakpoint is placed (to recover the instruction when the next break occurs)

With M command, set the trap instruction at the breakpoint (The trap instruction is 0xe7ffdefe)

Use c command for the "continue" operation

Materializing the step execution

The step execution with "s command" is processed by the stub as shown below

- How to use step execution exception
    - Step execution exception ... Exception that occurs after each instruction is executed
    - In many cases, the CPU register is changed to enable the exception for the step execution and continue the operation when s command is received
    - It is possible to cause a break for each instruction without rewriting instructions
    - In a sample stub for GDB, x86, IA-64, and 68000 are implemented this way.

Materializing the step execution

The step execution with "s command" is processed by the stub as shown below

- How to mount a trap instruction at the stub side
  - When s command is received, stubs save the next instruction of the program counter and rewrite it to the trap instruction to cause a break
  - For conditional branch instruction, trap instruction is necessary for both next to the branch instruction and where it branches to
  - When a break occurs, it is necessary to recover the saved instruction
  - In a sample stub for GDB, M32R and SH are implemented this way

Compression protocols

This can compress a sequence of 4 or more of same characters

Effective for operations such as setting registers using G command (By setting 0 for registers that don't require configuration, you can compress those values)

Format

```
<w>*<c>
```

`<w>`	a sequence of same characters
`<c>`	Using an expression n = c - 29, create additional n number of <w> in succession (when c = '<', n = '<' - 29 = 60 - 29 = 31)

Compression protocols

Example for ARM

```
#   R0      R1     R2-R13    LR      PC
#+------++------++-------++------++------+
G00240000000000000*<0*<0*<0000000024200000
```

a sequence of 32 zeros * 3 times -> 96 zeros
-> Cover the settings for 12 registers

When decompressed it looks like below

```
G002400000000000000000000000000000000000000000000...
...0000000024200000
```

Possibility of attacks by Step-Oriented Programming (SOP)

Is it really possible to do anything if the debugging port is stolen?

If the debugging port is stolen, an attacker can inject machine code using M command, modify the program counter using G command to point to the machine code, and then execute shell code easily (of course)

Is it really possible to do anything if the debugging port is stolen?

However, sometimes it is not possible to inject or execute machine code

- Possibilities of not being able to rewrite machine code
  - Guarded by memory protection
  - Stubs are checking addresses when writing into the memory using M command
  - Executable code are on the flash memory

- Possibilities of not being able to execute on data area
  - DEP is enabled
  - Hard to inject machine code due to small RAM
  - The microcomputer has the Harvard architecture that essentially does not allow machine code on RAM to be executed

Is it really possible to do anything if the debugging port is stolen?

Harvard Architecture

- From RISC's standpoint, having instruction cache separated from data cache
- From microcomputers' standpoint, having the address space of the flash ROM used to place the machine code separated from the address space of RAM used to place data (Ex: AVR)

(Essentially it is not possible to inject and execute machine code)

But, I don't say safety.

The principle of SOP

Even when it is not possible to inject and execute machine code, by repeatedly doing the following it is possible to combine existing machine code at instruction level and execute arbitrary machine code, similarly to ROP

- With G command, rewrite the program counter to the address of the desired machine code
- With s command, do the step execution

An arbitrary code execution based on a step execution by RSP (SOP: Step-Oriented Programming)

It's dangerous.

What you can do with SOP

- It is possible to combine existing machine code to execute arbitrary code, even if it is not possible to modify the machine code area or not possible to execute machine code on RAM

- It is possible to build arbitrary code more easily than methods like ROP because it can set the value of the program counter freely and execute instructions one by one using step execution

To do the step execution

It is possible to do the step execution, even when rewriting instructions is not possible

- Even if we can't rewrite the machine code, SOP is available without rewriting the machine code if the step execution uses step execution exception

- Some stubs have not been implemented to check addresses when a trap instruction is embedded by a step execution, despite having been implemented to check addresses when the memory is rewritten by M command

To do the step execution

It is possible to do the step execution, even when rewriting instructions is not possible

- Even when a step execution is not available, SOP is available by setting a breakpoint to the next instruction if the CPU has the hardware breakpoint function and the stubs support it (Z1 command)

- If the CPU has a watchpoint function, SOP is available by rewriting registers, combining a load instruction and a store instruction, and setting a breakpoint to an appropriate location

Experiments at SECCON CTF

Experiments through a question at SECCON CTF

For the online preliminary contest of SECCON2016, I experimentally presented a question for executing arbitrary code using SOP

For the final contest of SECCON2016, I presented a question to compete against each other to create the minimum SOP

- Executable files for 4 architectures such as ARM are operating on the simulator on the server
- Debugger's connection port is available and one can connect to it using RSP via TCP
- You get a score if you could read a keyword from a file "word.txt" in the current directory of the server and commit it to the score server
- You get a score at regular intervals if you write your team's keyword to a file "flag.txt" in the current directory of the server
  (Only the team with the minimum-sized SOP can write it)

Notes

The following 5 RSP commands are available: s, c, g, G, and m

M command is disabled

- Because there will be no place for creativity (the question will be so easy if you can rewrite the memory; you just inject and execute machine code)
- This assumes a situation where it is not possible to inject and execute machine code

SOP is required to complete this question
Is SOP used...?
Is SOP a method that anyone can come up with...?

Important!

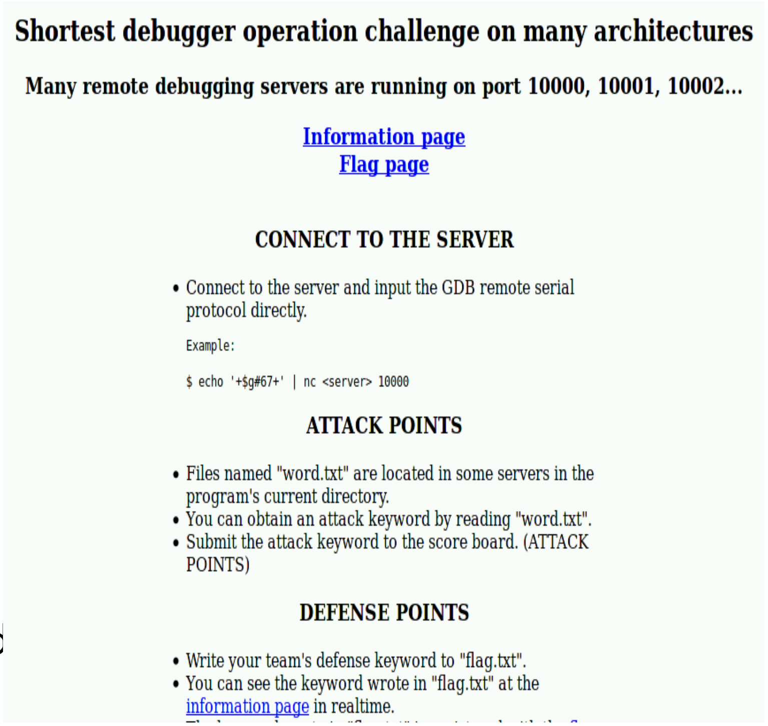
This research is conducted with the purpose that is to be able to consider some essential defenses by understanding the possibilities of attacks when the connection was taken away, not to help any illegal attacks.

It's illegal to attack or inspect others' server or equipment without permission. Don't do that!

Details of the presented question

Competitors get only the following information

- Port number to connect with TCP
- That you can connect using RSP
- That you score points by reading a keyword from word.txt
- That you score points by writing your team's keyword to flag.txt
- Example of a command to use after the connection is established ("+$g#67+")



Shortest debugger operation challenge on many architectures

Many remote debugging servers are running on port 10000, 10001, 10002...

Information page
Flag page

CONNECT TO THE SERVER

- Connect to the server and input the GDB remote serial protocol directly.

  Example:

  ```
  $ echo '+$g#67+' | nc <server> 10000
  ```

ATTACK POINTS

- Files named "word.txt" are located in some servers in the program's current directory.
- You can obtain an attack keyword by reading "word.txt".
- Submit the attack keyword to the score board. (ATTACK POINTS)

DEFENSE POINTS

- Write your team's defense keyword to "flag.txt".
- You can see the keyword wrote in "flag.txt" at the information page in realtime.

Key points in this question

- You must realize a precondition that you should operate directly from RSP, not from gdb
- You must find out what the architecture of the destination you are connecting to is
- You must search for commands available in RSP and understand which commands are available and which are not
- You must realize that the M command is unavailable
- You must think about how you can read from and write to files (Can they come up with the method to execute arbitrary code using SOP... ?)

As a result, many teams used SOP to solve the question (SOP is a method that anyone can come up with)

List of architectures

Architecture name	Type	Features
ARM	32-bit microcomputer	4-byte fixed-length instruction set
H8/300	16-bit microcontroller	Variable-length instruction set
SH	32-bit RISC microcontroller	2-byte fixed-length instruction set
V850	32-bit RISC microcontroller	4-byte fixed-length instruction set

There won't be a competition if there are many target servers, so we prepared only small number of architectures

Demo
(Connection using RSP)

Demo
(Obtaining keywords using SOP)

Results

As a result, many teams used SOP to solve the question

- Given a certain amount of knowledge, it can be said that SOP is a method that anyone can come up with (It isn't a special method, and it isn't a new method)

Use the existing code of the built-in library for floating-point arithmetic emulation of ARM simulator attached to GDB

- A variety of machine code becomes available
- I didn't know the existence of such a library

The team that is able to find and use the compression protocol for RSP will gain a huge advantage

Conclusion

- When the debugging port is open, anything can be done (of course)

- You cannot say that it is safe just because it is not possible to inject machine code
  (Using SOP, it is possible to execute arbitrary code)

- Let's make sure that the debugging port is closed!
  (If possible, try deleting stub code and removing the circuit itself)

Thank you
very much