black hat EUROPE 2020
DECEMBER 9-10
BRIEFINGS

Finding Bugs Compiler Knows but Doesn't Tell You: Dissecting Undefined Behavior Optimizations in LLVM

Zekai Wu (@hellowuzekai)

Wei Liu

Mingyue Liang (@MoonL1ang)

Kai Song (@ExpSky)

Who We Are

- Tencent Security Xuanwu Lab
  - Applied and real world security research
- About us: Members of Foundational Security Research Team
  - Zekai Wu(@hellowuzekai)
  - Wei Liu
  - Mingyue Liang(@MoonL1ang)
  - Kai Song(@ExpSky)

Tencent腾讯

腾讯安全玄武实验室
TENCENT SECURITY XUANWU LAB

Quiz

```
class A {
  public:
    void read(int x) {
      int *addr = internalRead(x);
      printf("0x%x\n", *addr);
    }
  private:
    int* internalRead(int x) {
      if (x < 0 || x >= 100){ return nullptr;}
      return array+x;
    }
    int flag = 0xdeadbeef;
    int array[100] = {0};
};
void main() {
      A a;  a.read(-1);
}
```

Nullptr dereference is "expected" if bound check fails

Array Bound Check

```
xxx@ubuntu:~$ clang++ demo.cpp -o demo
xxx@ubuntu:~$ ./demo
Segmentation fault (core dumped)
```

→ expected

```
xxx@ubuntu:~$ clang++ demo.cpp -O3 -o demo
xxx@ubuntu:~$ ./demo
0xdeadbeef
```

→ What???

Agenda

- Undefined Behavior in LLVM
- Undefined Behavior Detections
- Detection Result Case Study
- From Undefined Behavior to RCE
- Conclusions

Undefined Behavior

- Undefined Behavior(UB):
  - behavior upon use of a nonportable or erroneous program construct or of erroneous data
  - International Standard imposes no requirements
- C/C++ have lots of UB
  - C17 standard has 211 undefined behaviors
  - More new UBs will be added to the standard

Undefined Behavior

Undefined Behavior

Nullptr Dereference

```c
int *p = 0;
int b = *p + 1;
```

Oversized Shift

```c
int a = 1;
int b = a << 40;
```

Divide by 0

```c
int a = 1;
int b = a / 0;
```

Uninitialized Usage

```c
int a;
int b = a + 1;
```

Index Out of Bound

```c
int a[4] = {0};
int p = a[5];
```

Signed Integer Overflow

```c
int a = INT32_MAX;
int b = a + 1;
```

What is LLVM

- LLVM is a compiler infrastructure
  - LLVM provide high-quality libraries on code optimizations, analysis, code generator, profiling and debugging…
  - LLVM native compiler “clang” builds large number of software

UB in LLVM

- LLVM also has UB
  - “True” UB:
    - serious errors: divided by zero, illegal memory access
  - “Undef”:
    - come from uninitialized value
  - “Poison”:
    - support speculative executions.
    - converted into “True” UB when reaching a side-effecting instruction

When LLVM meet UB

- What happens when LLVM meet undefined behavior
  - program works as expected
  - fail to compile
  - unpredictable or nonsensical result
  - memory corruption

- Security threat of UB:
  - Direct memory corruptions:
    - UB such as array index out-of-bound error lead to memory corruptions
  - Program semantics may be changed unexpectedly
    - Sanity check may be removed if it contains UB code
- We need to find ways to detect undefined behavior

Agenda

- Undefined Behavior in LLVM
- Undefined Behavior Detections
- Detection Result Case Study
- From Undefined Behavior to RCE
- Conclusions

Existing Detection

- Existing detection method of Undefined Behavior:
  - Dynamic Analysis: UBSAN, ASAN, MSAN, TSAN
  - Static Analysis: Clang Static Analyzer, Coverity, Frame-C/TIS Analyzer

Static Detection

- Dynamic Analysis:
  - Undefined Behavior Sanitizer(UBSAN): Shift errors, signed integer overflow
  - Address Sanitizer(ASAN): Memory safety errors
  - Memory Sanitizer(MSAN): Use of uninitialized variable
  - Thread Sanitzer(TSAN): Data races, deadlocks
  - They all need test cases to trigger bugs

Static Detection

- Static Analysis:
  - Enable existing compiler warnings: -Werror
  - Static analysis tools: Clang Static Analyzer, Coverity …
  - Only detect a fraction of UB
  - Don’t make the best use of compilers’ analysis on programs

UB Branch Elimination Case

Any simple but effective ways to detect UB?

Quiz Analysis

```
class A {
  public:
    void read(int x) {
      int *addr = internalRead(x);
      printf("0x%x\n", *addr);
    }
 private:
    int* internalRead(int x) {
      if (x < 0 || x >= 100){ return nullptr;}
      return array+x;
    }
    int flag = 0xdeadbeef;
    int array[100] = {0};
};
int main() {
      A a;  a.read(-1); return 1;
}
```

```
xxx@ubuntu:~$ clang++ demo.cpp -O3 -o demo
xxx@ubuntu:~$ ./demo
0xdeadbeef
```

No bound check branch in function “main”

```
0000000000400580 <main>:
  400580: 50                      push   %rax
  400581: bf 24 06 40 00          mov    $0x400624,%edi
  400586: be ef be ad de          mov    $0xdeadbeef,%esi
  40058b: 31 c0                   xor    %eax,%eax
  40058d: e8 de fe ff ff          callq  400470
<printf@plt>
  400592: b8 01 00 00 00          mov    $0x1,%eax
  400597: 59                      pop    %rcx
  400598: c3                      retq
```

Quiz Analysis

```
class A {
  public:
    void read(int x) {
      int *addr = internalRead(x);
      printf("0x%x\n", *addr);
    }
private:
    int* internalRead(int x) {
      if (x < 0 || x >= 100){
         return nullptr;}
      return array+x;
    }
    int flag = 0xdeadbeef;
    int array[100] = {0};
};
int main() {
      A a;  a.read(-1); return 1;
}
```

xxx@ubuntu:~$ clang++ demo.cpp -O3 -o demo -mllvm –print-after-all

function “Read” at early stage of compilation:

```
void @_ZN1A4readEi(%class.A* %0, i32 %1) {
  %3 = call i32* @_ZN1A12internalReadEi(%class.A* %0, i32 %1)
  %4 = load i32, i32* %3, align 4, !tbaa !7
  %5 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds
       ([6 x i8], [6 x i8]* @.str, i64 0, i64 0), i32 %4)
  ret void
}
```

Bound check is missing in function “read”

function “Read” before Inlined to function “main”:

```
void @_ZN1A4readEi(%class.A* %0, i32 %1) {
  %3 = sext i32 %1 to i64
  %4 = getelementptr inbounds %class.A, %class.A* %0, i64
       0, i32 1, i64 %3
  %5 = load i32, i32* %4, align 4, !tbaa !7
  %6 = tail call i32 (i8*, ...) @printf(i8* nonnull
       dereferenceable(1) getelementptr inbounds ([6 x i8],
       [6  x i8]* @.str, i64 0, i64 0), i32 %5)
  ret void
}
```

Quiz Analysis

```cpp
class A {
  public:
    void read(int x) {
      int *addr = internalRead(x);
      printf("0x%x\n", *addr);
    }
private:
    int* internalRead(int x) {
      if (x < 0 || x >= 100){
         return nullptr;}
      return array+x;
    }
    int flag = 0xdeadbeef;
    int array[100] = {0};
};
int main() {
      A a;  a.read(-1); return 1;
}
```

xxx@ubuntu:~$ clang++ demo.cpp -O3 -o demo -mllvm –print-after-all

```
*** IR Dump Before Combine redundant instructions ***
void @_ZN1A4readEi(%class.A* %0, i32 %1) {
  %3 = icmp ugt i32 %1, 99
  %4 = sext i32 %1 to i64
  %5 = getelementptr inbounds %class.A, %class.A* %0, i64 0,
i32 1, i64 %4
  %6 = select i1 %3, i32* null, i32* %5
  %7 = load i32, i32* %6, align 4, !tbaa !7
  %8 = call i32 (i8*, ...) @printf(i8* nonnull
dereferenceable(1) getelementptr inbounds ([6 x i8], [6 x
i8]* @.str, i64 0, i64 0), i32 %7)
}

*** IR Dump After Combine redundant instructions ***
void @_ZN1A4readEi(%class.A* %0, i32 %1) {
  %3 = sext i32 %1 to i64
  %4 = getelementptr inbounds %class.A, %class.A* %0, i64
0, i32 1, i64 %3
  %5 = load i32, i32* %4, align 4, !tbaa !7
  %6 = call i32 (i8*, ...) @printf(i8* nonnull
dereferenceable(1) getelementptr inbounds ([6 x i8], [6 x
i8]* @.str, i64 0, i64 0), i32 %5)
}
```

Quiz Analysis

```
//file:llvm/lib/Transforms/InstCombine/InstCombineLoadStoreAlloca.cpp
Instruction  *InstCombiner::visitLoadInst(
LoadInst &LI) {
...
// load (select (Cond, &V1, &V2)) -->
     select (Cond, load &V1, load &V2).
if (SelectInst *SI = dyn_cast<SelectInst>(Op)){
  ...
  // load (select (cond, null, P)) -> load P
  if (isa<ConstantPointerNull>(SI->getOperand(1))
    && !NullPointerIsDefined(SI->getFunction(),
       LI.getPointerAddressSpace()))
   return replaceOperand(LI, 0, SI->getOperand(2));
```

```
*** IR Dump Before Combine redundant instructions ***
void @_ZN1A4readEi(%class.A* %0, i32 %1) {
  %3 = icmp ugt i32 %1, 99
  %4 = sext i32 %1 to i64
  %5 = getelementptr inbounds %class.A, %class.A* %0,
i64 0, i32 1, i64 %4
  %6 = select i1 %3, i32* null, i32* %5
  %7 = load i32, i32* %6, align 4, !tbaa !7
  %8 = call i32 (i8*, ...) @printf(i8* nonnull
dereferenceable(1) getelementptr inbounds ([6 x i8],
[6 x i8]* @.str, i64 0, i64 0), i32 %7)
}
```

- LLVM found “Nullptr Load” UB bug, but just fold it
- Uses of “Nullptr Load” including bound check comparison is removed
- Then an OOB access vulnerability appears

Thinkings

- LLVM can find undefined behavior when compiling programs:
  - LLVM won't tell programmers they found UB bug
  - LLVM tends to optimize UB code and sometimes even creates security vulnerabilities programmers don't know
- Why not just use LLVM's findings to detect undefined behavior?

Our Detection

```
//file:llvm/lib/Transforms/InstCombine/InstCombineLoadSto
reAlloca.cpp
Instruction  *InstCombiner::visitLoadInst(
LoadInst &LI) {
...
// load (select (Cond, &V1, &V2)) -->
select (Cond, load &V1, load &V2).
if (SelectInst *SI = dyn_cast<SelectInst>(Op)){
  ...
  // load (select (cond, null, P)) -> load P
  if (isa<ConstantPointerNull>(SI->getOperand(1))
   && !NullPointerIsDefined(SI->getFunction(),
       LI.getPointerAddressSpace()))
   // add hooks
   return replaceOperand(LI, 0, SI->getOperand(2));
```

```
outs() << "load (select (cond, null, P)) -> load
P\n";
outs() << " Inst: " << *LI << "\n";
DebugLoc dl = LI->getDebugLoc();
if (dl){
    dl.print(outs());
}
outs() << "\n";
```

- Add hooks to track where “Nullptr Load” UB happens
- Limited, only “Nullptr Load” in “select” instruction can be found

Our Detection

- Add more hooks to find UB from LLVM's program analysis:
  - Challenges of locating where to instrument:
    - LLVM has a large code base of over 6 million lines
    - "True" UB, "Undef" and "Poison" are mixed up together
    - We prefer UB bugs that has security impact

Our Detection

- Add more hooks to find UB from LLVM’s program analysis:
  - Combine source review with manual experiments:
  - Look for UB that has security threat:
    - Focus on “UB” optimizations that may change program semantics
    - Focus on “UB” bugs that are security vulnerabilities themselves

Our Detection

- Folding UB in “Select” instruction may change program semantics

```
//file:llvm/lib/Transforms/InstCombine/InstCombineLoadStoreAlloca.cpp
Instruction  *InstCombiner::visitLoadInst(
LoadInst &LI) {
if (SelectInst *SI = dyn_cast<SelectInst>(Op)){
  ...
  // load (select (cond, null, P)) -> load P
  if (isa<ConstantPointerNull>(SI-
>getOperand(1))
    && !NullPointerIsDefined(SI->getFunction(),
        LI.getPointerAddressSpace()))
```

```
// file: llvm/lib/Transforms/Scalar/SCCP.cpp
void SCCPSolver::visitSelectInst(SelectInst &I) {
...
if (TVal.isUnknown()) // select ?, undef, X -> X.
return (void)mergeInValue(&I, FVal);
if (FVal.isUnknown()) // select ?, X, undef -> X.
return (void)mergeInValue(&I, TVal);
```

```
// file: llvm/lib/IR/ConstantFold.cpp
Constant
*llvm::ConstantFoldSelectInstruction(Constant
*Cond, Constant *V1, Constant *V2) {
  ...
  if (isa<UndefValue>(Cond)) {
    if (isa<UndefValue>(V1)) return V1;
      return V2;
  }
  if (isa<UndefValue>(V1)) return V2;
  if (isa<UndefValue>(V2)) return V1;
```

```
// file: llvm/lib/Analysis/InstructionSimplify.cpp
static Value *SimplifySelectInst(Value *Cond, Value
*TrueVal, Value *FalseVal,
  const SimplifyQuery &Q, unsigned MaxRecurse) {
  ...
  if (isa<UndefValue>(TrueVal))
    return FalseVal; // select ?, undef, X -> X
  if (isa<UndefValue>(FalseVal))
    return TrueVal;  // select ?, X, undef -> X
```

Our Detection

- “Branch” instruction containing UB will be removed

```
//file: llvm/lib/Transforms/Utils/SimplifyCFG.cpp
static bool removeUndefIntroducingPredecessor(BasicBlock *BB) {
  for (PHINode &PHI : BB->phis())
    for (unsigned i = 0, e = PHI.getNumIncomingValues(); i != e; ++i)
      if (passingValueIsAlwaysUndefined(PHI.getIncomingValue(i), &PHI)) {
        Instruction *T = PHI.getIncomingBlock(i)->getTerminator();
        IRBuilder<> Builder(T);
        if (BranchInst *BI = dyn_cast<BranchInst>(T)) {
          BB->removePredecessor(PHI.getIncomingBlock(i));
```

Our Detections

- Find “UB” bugs that are security vulnerabilities themselves:
  - Select appropriate type of UB:
    - Array index OOB bug is attractive but hard to model
    - Integer Overflow/Uninitialized Usage are great candidates
  - Vulnerable test cases help us navigate to LLVM’s UB handling code

Integer Overflow Case

```
void test(int size) {
    // Detect integer overflow UB
    if (size > size+1){
        printf("Size Overflow!\n");
        return;
    }
    int a = size + 1;
    printf("Size: %d\n", a);
}
int main(int argc, char** argv){
    test(INT32_MAX);
    return 0;
}
```

Integer overflow sanity checks will be removed

```
xxx@ubuntu:~$ clang++ demo.cpp -o demo
xxx@ubuntu:~$ ./demo
Size Overflow!
xxx@ubuntu:~$ clang++ demo.cpp -O3 -o demo
xxx@ubuntu:~$ ./demo
Size: -2147483648
```

Integer Overflow Detection

```
// file: llvm/lib/Analysis/InstructionSimplify.cpp
static Value *simplifyICmpWithBinOp(CmpInst::Predicate Pred, Value *LHS,
                                    Value *RHS, const SimplifyQuery &Q,
                                    unsigned MaxRecurse) {
  Type *ITy = GetCompareTy(LHS); // The return type.
  BinaryOperator *LBO = dyn_cast<BinaryOperator>(LHS);
  BinaryOperator *RBO = dyn_cast<BinaryOperator>(RHS);
  if (MaxRecurse && (LBO || RBO)) {
    ...
    // Analyze the case when either LHS or RHS is an add instruction.
    // LHS = A + B (or A and B are null); RHS = C + D (or C and D are null).
    // icmp (X+Y), X -> icmp Y, 0 for equalities or if there is no overflow.
    if ((A == RHS || B == RHS) && NoLHSWrapProblem)
      if (Value *V = SimplifyICmpInst(Pred, A == RHS ? B : A,
            Constant::getNullValue(RHS->getType()), Q,MaxRecurse - 1))
        return V;
```

Instrumentation here will find potential integer overflow bugs

Integer Overflow Case

```
uint64_t test(uint16_t x, uint16_t y)
{
  size_t i = 15;
  uint64_t a = 0;
  a |= (((x) << (2*i)) | ((y) << (2*i + 2)));
  return a;
}

int main(int argc, char **argv)
{
  printf("test(0, 0) = %lu\n", test(0, 0));
  return 0;
}
```

0 | Undef

1. Tracking undef binary operation helps finding potential overflow bugs
2. Constant folding undef sometimes returns an abnormal value

```
xxx@ubuntu:~$ clang++ demo.cpp -o demo
xxx@ubuntu:~$ ./demo
test(0, 0) = 0x0
xxx@ubuntu:~$ clang++ demo.cpp -O3 -o demo
xxx@ubuntu:~$ ./demo
test(0, 0) = 0xffffffffffffffff
```

Integer Overflow Detection

```
//file: llvm/lib/IR/ConstantFold.cpp
Constant *llvm::ConstantFoldBinaryInstruction(unsigned Opcode, Constant *C1, Constant *C2)
{
  ...
  bool HasScalarUndefOrScalableVectorUndef =
    (!C1->getType()->isVectorTy() || IsScalableVector) &&
    (isa<UndefValue>(C1) || isa<UndefValue>(C2));
  if (HasScalarUndefOrScalableVectorUndef) {
  ...
    case Instruction::Or: // X | undef -> -1
      if (isa<UndefValue>(C1) && isa<UndefValue>(C2))
        // undef | undef -> undef
        return C1;
      return Constant::getAllOnesValue(C1->getType());
      // undef | X -> ~0
```

Instrumentation here helps finding undef

Instrumentation here helps finding tracking abnormal constant folding result

Our Detection

- Filter false positives:
  - distinguish false positives brought by “Poison” UB
  - abandon cases where we cannot control input to trigger UB side effect

Our Detection

- Summary: pick up UB found by LLVM
  - Dig into LLVM internals to figure out LLVM’s UB handling code
  - Add instrumentations to log the UB info found by LLVM
  - Use hooked clang to compile programs to find UB bugs
  - Filter false positives and construct PoC to trigger bugs
- We scan chromium, android AOSP with our “UB” detection tools

Agenda

- Undefined Behavior in LLVM
- Undefined Behavior Detections
- Detection Result Case Study
- From Undefined Behavior to RCE
- Conclusions

CVE-2020-0452

UB

```c
const char *
exif_entry_get_value(ExifEntry *e, char *val, unsigned int maxlen){
  ...
  case EXIF_TAG_XP_SUBJECT:
  {
    if (e->size+sizeof(unsigned short) < e->size) break;
    unsigned short *utf16 = exif_mem_alloc (e->priv->mem,
                                            e->size+sizeof(unsigned short));
    if (!utf16) break;
    memcpy(utf16, e->data, e->size);
    utf16[e->size/sizeof(unsigned short)] = 0;
    exif_convert_utf16_to_utf8(val, utf16, maxlen);
    exif_mem_free(e->priv->mem, utf16);
    break;
  }
```

Not Only CVE-2020-0452

```
if ((doff + s < doff) || (doff + s < s) ||
(doff + s > size)) {
    exif_log (data->priv->log,
        EXIF_LOG_CODE_DEBUG, "ExifData",
        "Tag data past end of buffer (%u > %u)",
        doff+s, size);
    return 0;
}
```

CVE-2019-9278

```
if ((o + s < o) || (o + s < s) || (o + s > ds)
|| (o > ds)) {
    exif_log (data->priv->log,
    EXIF_LOG_CODE_DEBUG, "ExifData",
        "Bogus thumbnail offset(%u) or size(%u)",
        o, s);
    return;
}
```

CVE-2020-0181

```
if ((offset + 2 < offset)||(offset + 2 < 2)
|| (offset + 2 > ds)) {
    exif_log (data->priv->log,
    EXIF_LOG_CODE_CORRUPT_DATA, "ExifData",
    "Tag data past end of buffer (%u > %u)",
    offset+2, ds);
    return;
}
```

CVE-2020-0198

```
if ((datao + 2 < datao) || (datao + 2 < 2) ||
(datao + 2 > buf_size)) {
    exif_log (ne->log,
    EXIF_LOG_CODE_CORRUPT_DATA,
    "ExifMnoteCanon", "Short MakerNote");
    return;
}
```

CVE-2020-13112

Not Only CVE-2020-0452

- Using undefined behavior to do sanity check is a popular programming paradigm and works well in old compilers
- But they lead to vulnerabilities in modern heavily optimized compilers like clang
- These old libs are still widely used (eg: libexif was first released in 2002, but is still integrated in Android media framework)

CVE-2020-6523

```
sk_sp<GrTextBlob> GrTextBlob::Make(...) {
  ...
  size_t vertexToSubRunPadding = alignof(SDFT3DVertex) -alignof(SubRun);
  size_t arenaSize = sizeof(GrGlyph*) * glyphRunList.totalGlyphCount()
    + quadSize * glyphRunList.totalGlyphCount()
    + glyphRunList.runCount() * (sizeof(SubRun) + vertexToSubRunPadding);

  size_t allocationSize = sizeof(GrTextBlob) + arenaSize;

  void* allocation = ::operator new (allocationSize);
  ...
}
```

Undef

UB

crbug 1071311

```
void set(int index) {
    uint32_t mask = 1 << (index & 31);
    uint32_t* chunk = this->internalGet(index);
    SkASSERT(chunk);
    *chunk |= mask;
}
```

UB

```
uint32_t* internalGet(int index) const {
    size_t internalIndex = index / 32;
    if (internalIndex >= fDwordCount) {
        return nullptr;
    }
    return fBitData.get() + internalIndex;
}
```

OOB

crbug 1076387

```
WebGLTimerQueryEXT::WebGLTimerQueryEXT(WebGLRenderingContextBase* ctx)
    : WebGLContextObject(ctx),
      ...
      task_runner_(
          ctx->canvas()
             ->GetDocument()
             .GetTaskRunner(TaskType::kInternalDefault)) {
  Context()->ContextGL()->GenQueriesEXT(1, &query_id_);
}
```

UB

```
HTMLCanvasElement* canvas() const {
    if (Host()->IsOffscreenCanvas())
        return nullptr;
    return static_cast<HTMLCanvasElement*>(Host())
}
```

Type Confusion

Agenda

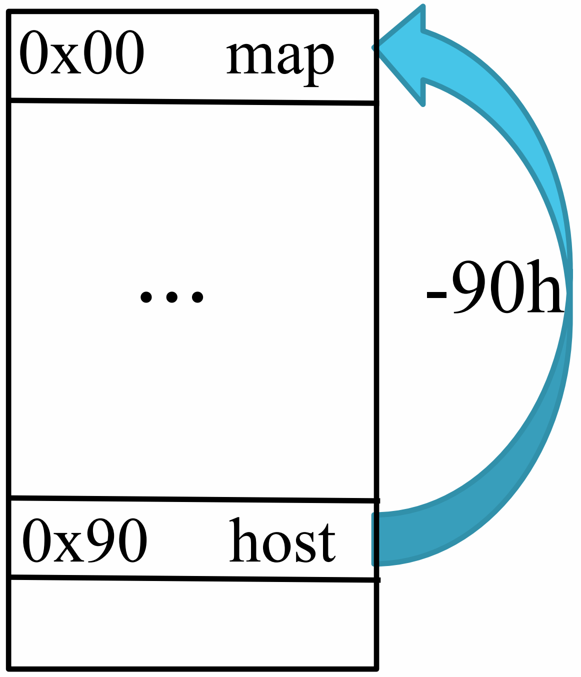
- Undefined Behavior in LLVM
- Undefined Behavior Detections
- Detection Result Case Study
- From Undefined Behavior to RCE
- Conclusions

crbug 1076387 exploit

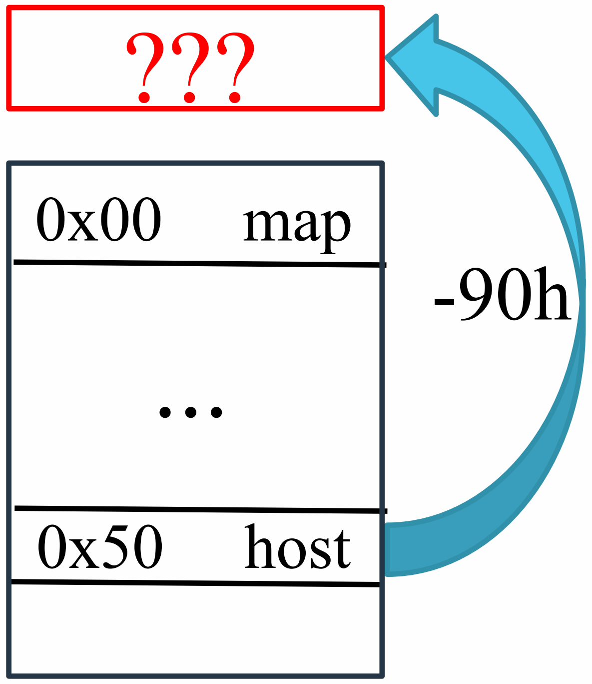
```
HTMLCanvasElement* canvas() const {
    if (Host()->IsOffscreenCanvas())
        return nullptr;
    return static_cast<HTMLCanvasElement*>(Host());
}
```

```
mov rax,qword ptr [rbx+20h]; host()
lea rcx,[rax-90h]           ; static_cast
```

ASM Code of canvas()



HTMLCanvasElement



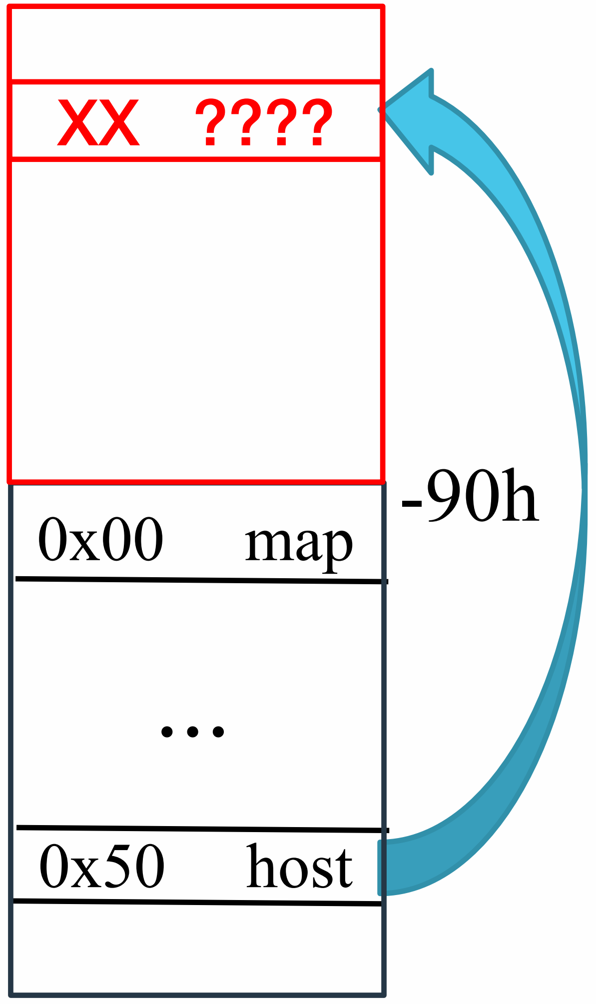
OffscreenCanvas

crbug 1076387 exploit

```
WebGLTimerQueryEXT::WebGLTimerQueryEXT(WebGLRenderingContextBase* ctx)
    : WebGLContextObject(ctx),
      ...
      task_runner_(
          ctx->canvas()
             ->GetDocument()
             .GetTaskRunner(TaskType::kInternalDefault))
```

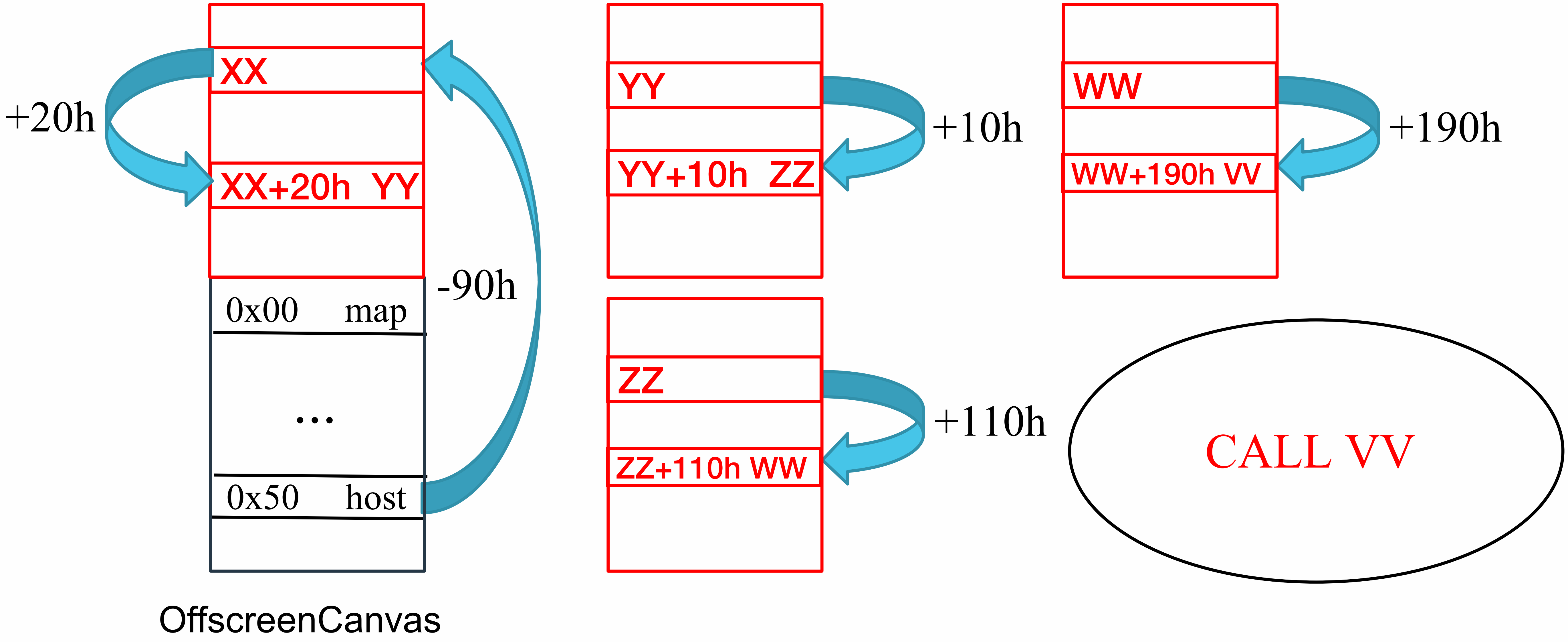
```
mov     rax,qword ptr [rbx+20h] ; host()
lea     rcx,[rax-90h]           ; static_cast
mov     rax,qword ptr [rcx+20h]
mov     rcx,qword ptr [rax+10h]
mov     rax,qword ptr [rcx+110h]
call    qword ptr [rax+190h]
```

ASM code of inline function



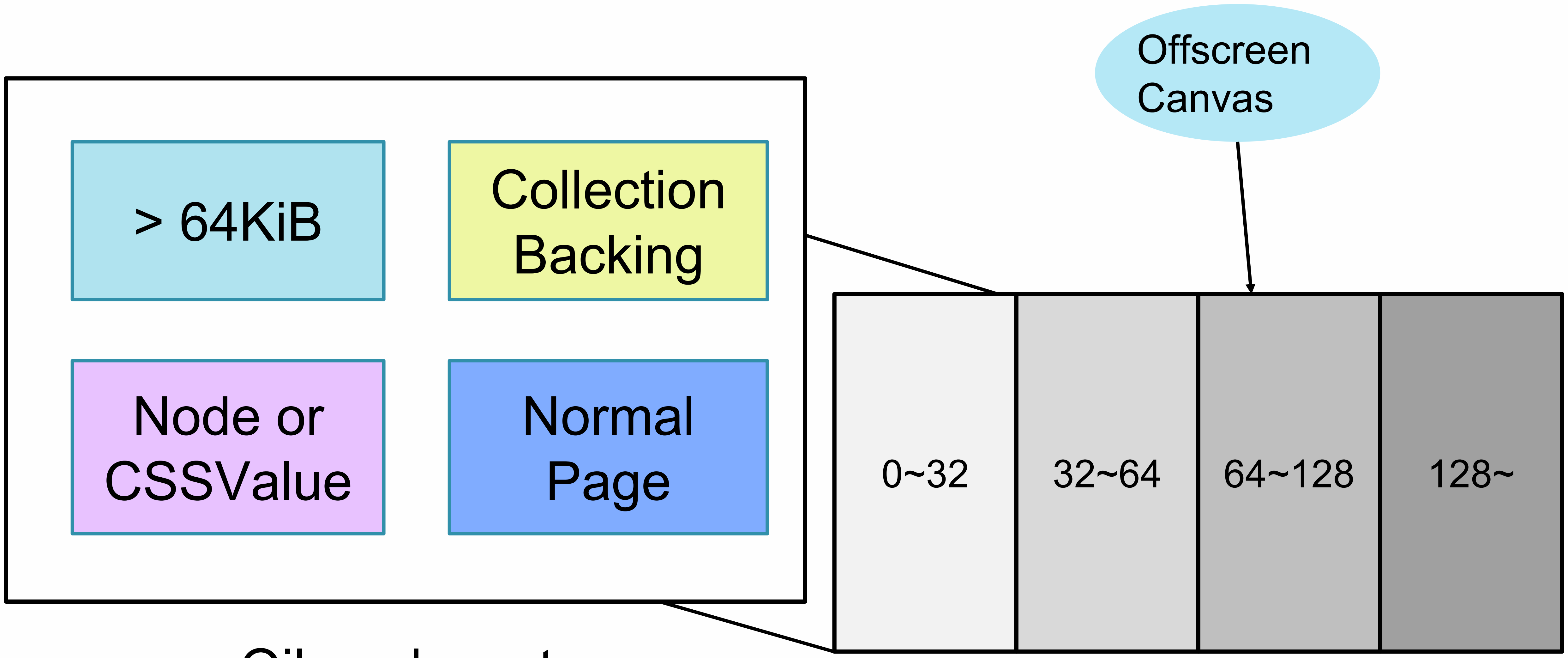
OffscreenCanvas

crbug 1076387 exploit



Oilpan

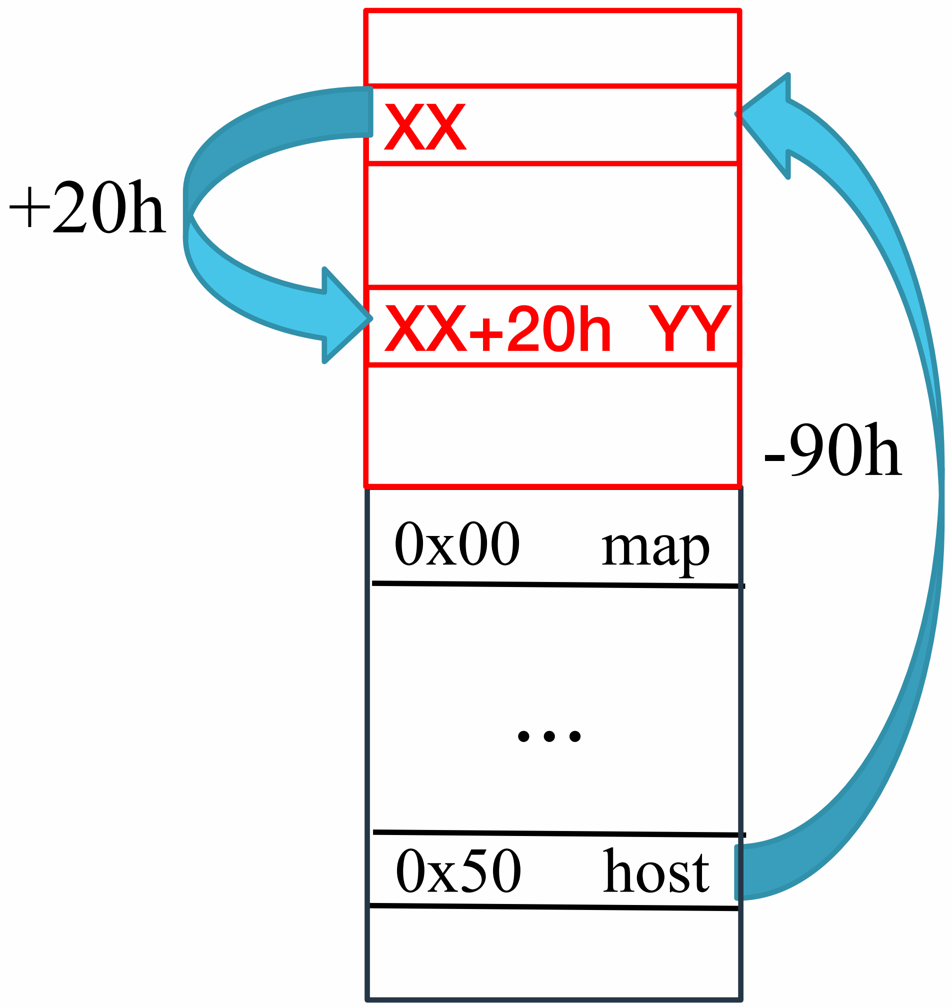
- OffscreenCanvas is a Garbage Collected (GC) object in blink
- Blink use Oilpan to manage GC object



Oilpan layout

Search Object

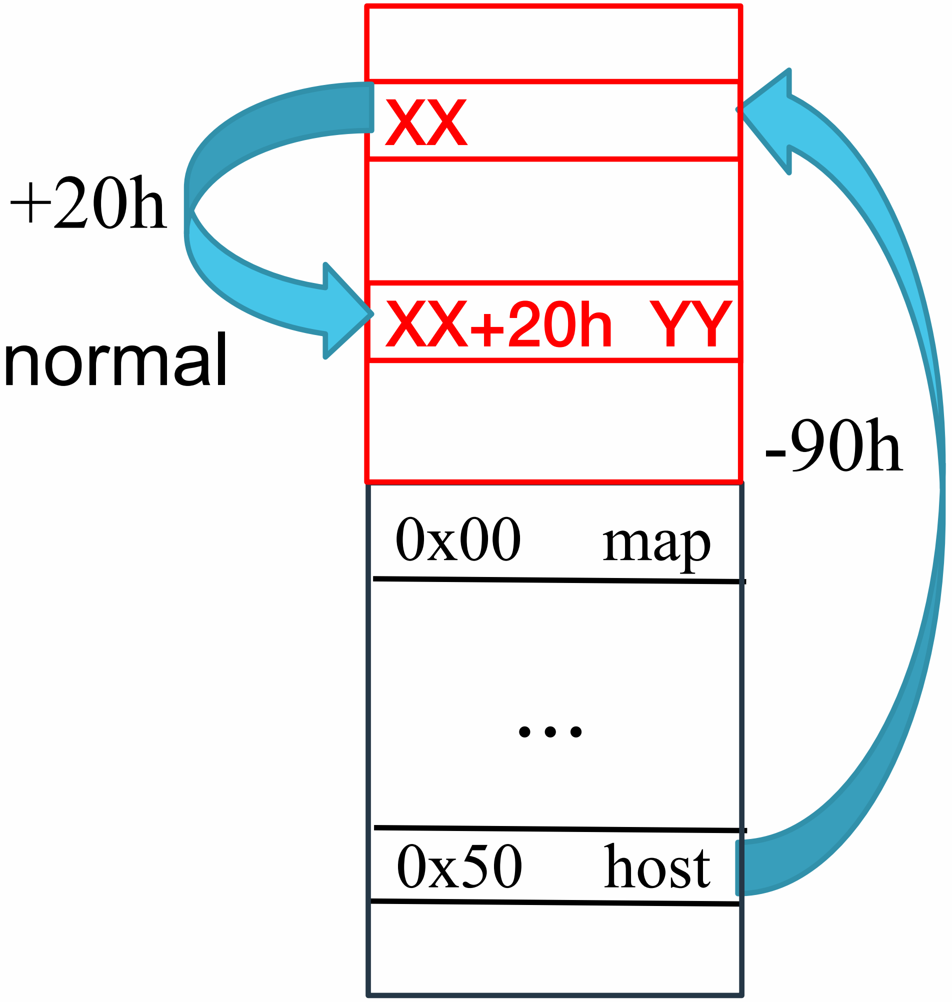
- Requirement of suitable objects
  - Located at the 64 ~ 128 bytes bucket of Normal page Arenas
  - Value of YY can be controlled



OffscreenCanvas

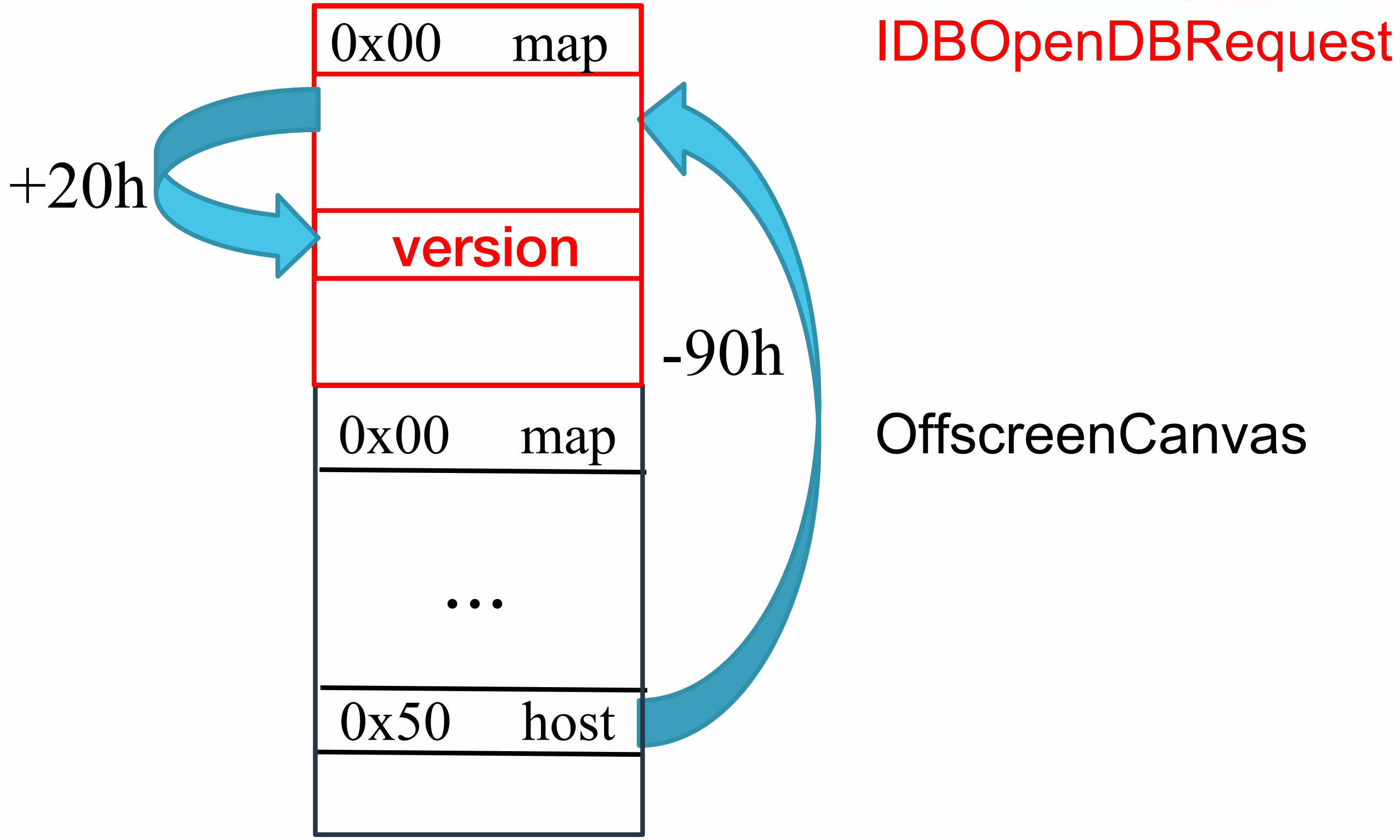
Search Method

- Search Method
  - CodeQL
    - Find object is allocated by Oilpan and located at normal page arena
    - Find object size between 64 and 128 bytes
  - Code Review
    - Ensure value of YY can be controlled



OffscreenCanvas

Control Register



window.indexedDB
.open(”t",0x123456789)

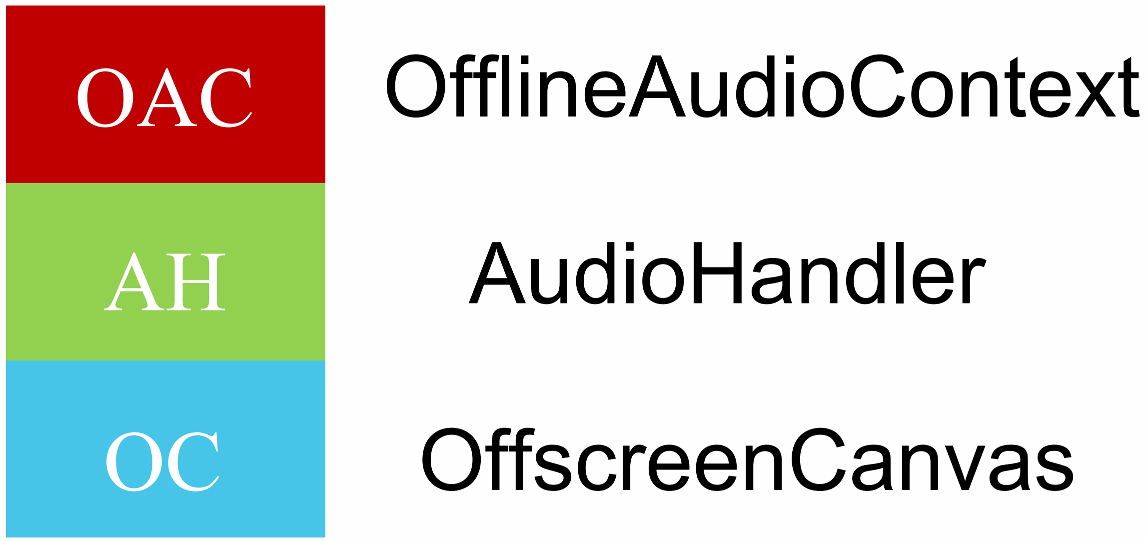
(fbc.29c0): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
*** WARNING: Unable to verify checksum for D:\chrome-win\chrome_child.dll
chrome_child!blink::MemberBase [inlined in chrome_child!blink::WebGLTimerQueryEXT::WebGLTimerQueryEXT+0x5b]:
00007fff`4f991a63 488b4810 mov rcx,qword ptr [rax+10h] ds:00000001`23456799=????????????????

Exploit 32-bit Chrome

- Exploit 32-bit Chrome
  - 32-bit chrome has a smaller memory address space
  - Heap Spray technique can make it easier to control EIP

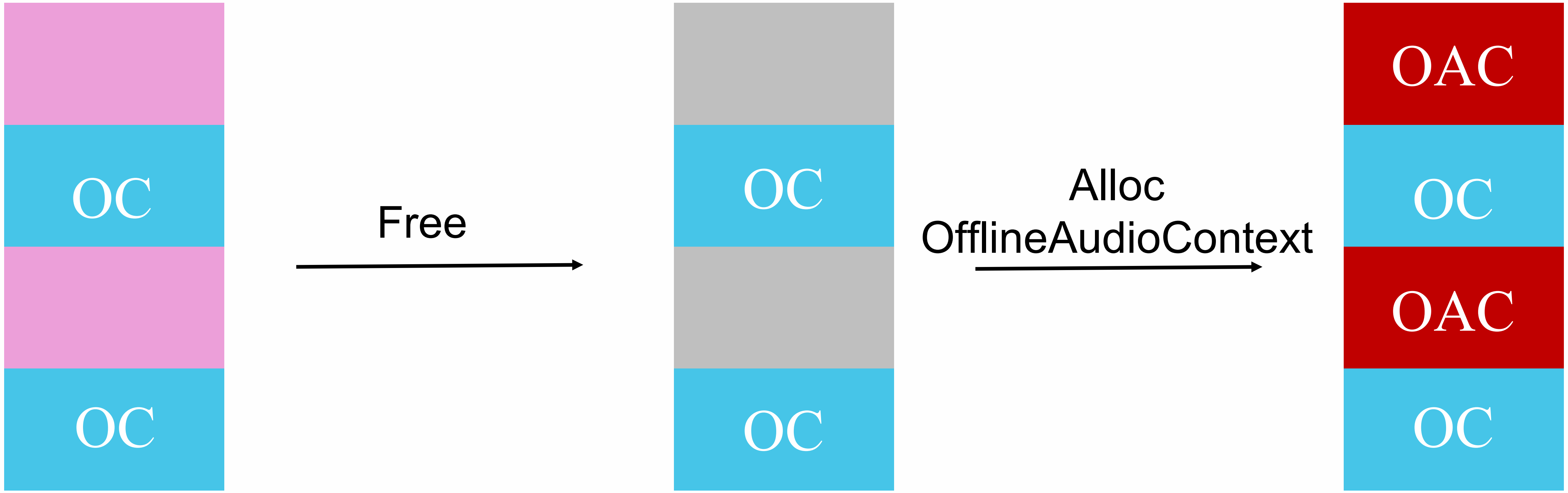
Search Object

- Search Object on 32-bit Chrome
  - OfflineAudioContext.length in the right place
  - But creating an OfflineAudioContext will create an AudioHandler at the same time

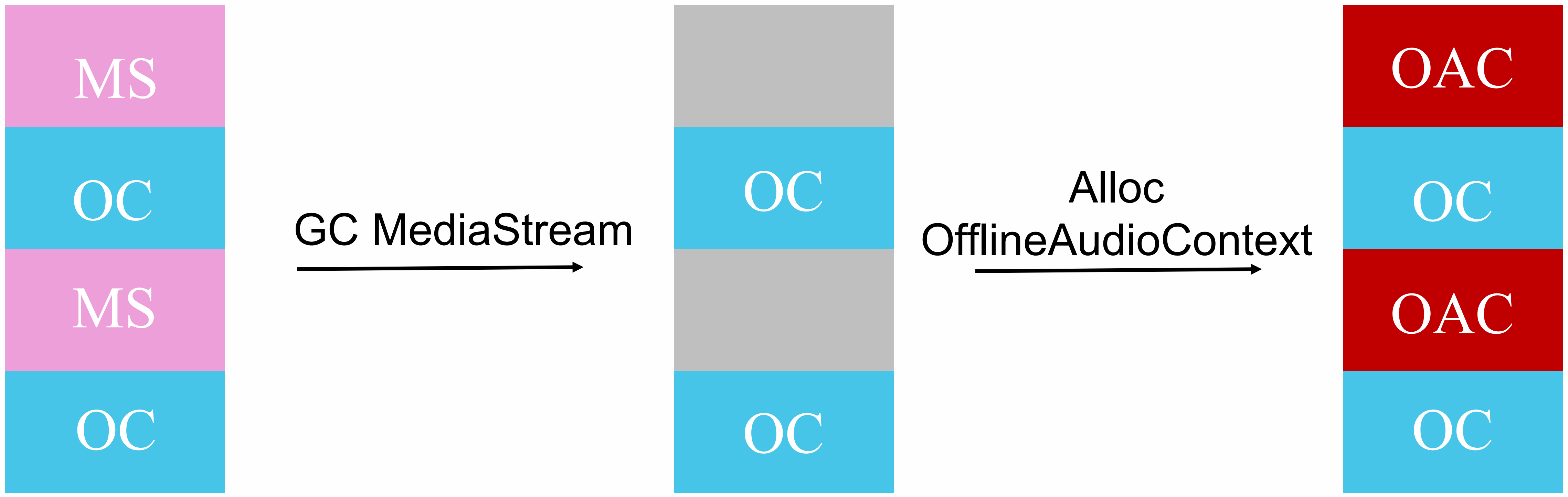


Heap Fengshui

- Oilpan uses freelist to manage freed memory
  - Create an object of the same size as AOC next to OC
  - Free it and now this memory is managed by the freelist
  - Create OAC, it will use the previously freed memory



Control Register

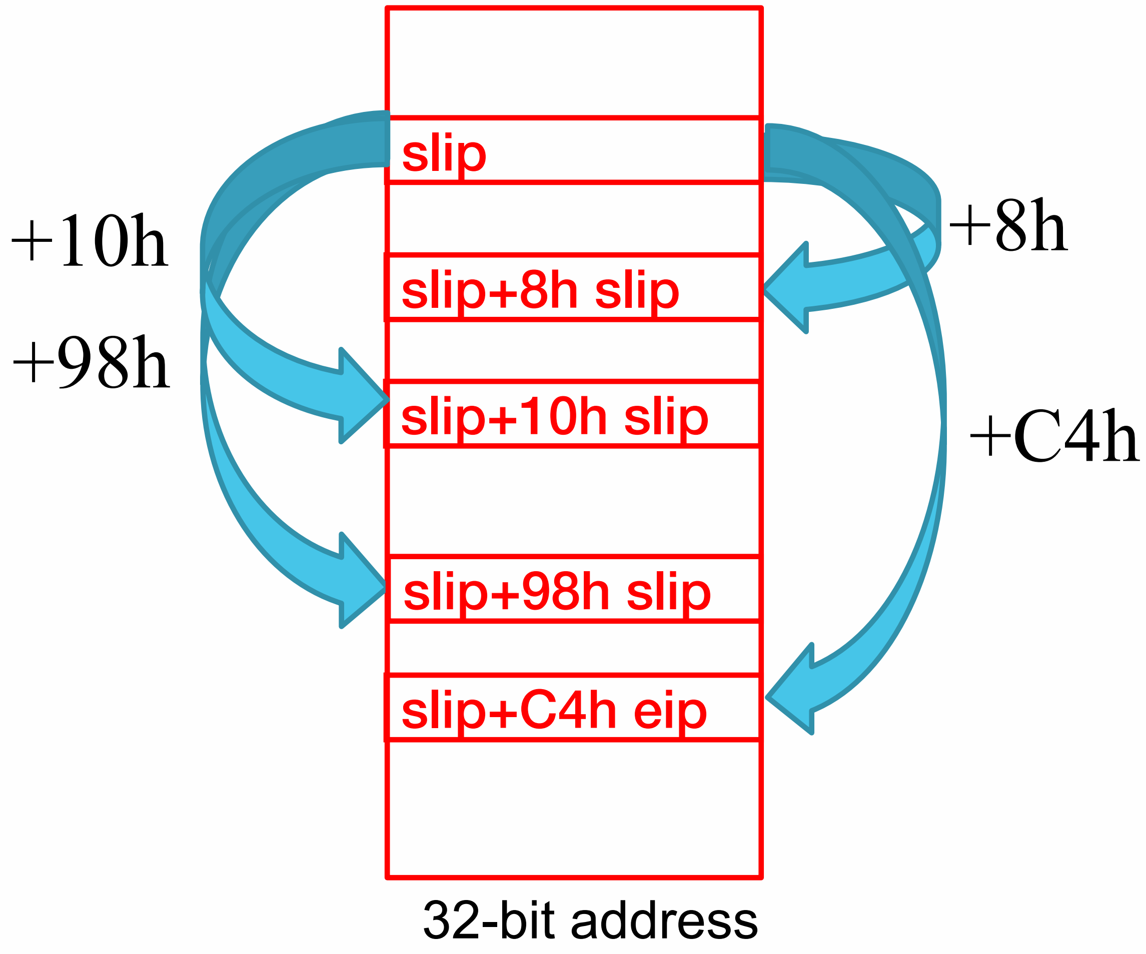


sizeof(MediaStream) = sizeof(OfflineAudioContext)

```
Breakpoint 0 hit
eax=42dcbc5c ebx=17bd7bd8 ecx=42dcbc14 edx=00010003 esi=17a84610 edi=17a84640
eip=5e724544 esp=007fe65c ebp=007fe66c iopl=0         nv up ei pl nz na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00200206
chrome_child!blink::MemberBase<blink::TreeScope,blink::TracenessMemberConfiguration::kTraced>::Ge
tRaw [inlined in chrome_child!blink::WebGLTimerQueryEXT::WebGLTimerQueryEXT+0x4c]:
5e724544 8b4110          mov     eax,dword ptr [ecx+10h] ds:002b:42dcbc24=12345678
```

Heap Spray

- Heap Spray on 32bit Chrome
  - Design the structure of spraying chunk
  - Allocate large number of chunks
  - Set slip to ecx+10




```
(1404.2eb8): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
*** WARNING: Unable to verify checksum for D:\chrome-win\chrome.dll
eax=110cf000 ebx=8dee9270 ecx=110cf098 edx=8ba04000 esi=8e041d18 edi=8e041d48
eip=12345678 esp=0077ea58 ebp=0077ea74 iopl=0         nv up ei pl nz na po nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00210202
12345678 c23708          ret     837h
```

```
mov     eax,dword ptr [ecx+10h]
mov     ecx,dword ptr [eax+8]
mov     eax,dword ptr [ecx+98h]
call    dword ptr [eax+0C4h]
```

JIT Spraying

- JIT Spraying:
  - JIT spraying circumvents the protection of ASLR and DEP by exploiting the behavior of just-in-time compilation.
  - The purpose of JIT is to produce executable data.
  - The input program typically contains numerous constant values that can be erroneously executed as code.

JIT Spraying

js code

```
var a = (0x11223344 ^ 0x90909090 ^ 0x90909090);
```

jit code

```
0: b8 44 33 22 11     mov eax,0x11223344
5: 35 90 90 90 90     xor eax,0x90909090
a: 35 90 90 90 90     xor eax,0x90909090
```

jit code with offset

```
1: 44                 inc esp
2: 33 22              xor esp,DWORD PTR [edx]
4: 11 35 90 90 90 90  adc DWORD PTR ds:0x90909090,esi
a: 35 90 90 90 90     xor eax,0x90909090
```

JIT Spraying

js code

```
var a = (0x11223344 ^ 0xa8909090 ^ 0xa8909090);
```

jit code

```
0: b8 44 33 22 11      mov eax,0x11223344
5: 35 90 90 90 a8      xor eax,0xa8909090
a: 35 90 90 90 a8      xor eax,0xa8909090
```

jit code with offset

```
9: a8 35               test  al,0x35
b: 90                  nop
c: 90                  nop
d: 90                  nop
```

3 bytes in every 5 bytes can be used encode instruction

JIT Spraying

- Tricks
  - The register used by XOR is random.

```
9: a8 35        test   al,0x35
b: 90           nop
c: 90           nop
d: 90           nop

    35 ;xor eax
```

```
9: a8 83        test   al,0x83
b: f1           ???
c: 90           nop
d: 90           nop

    83 f1 ;xor ecx
```

  - Change the format of the xor statement to adjust the registers it uses

```
a ^ 0xa8909090   =>   a ^ b ^ 0xa8909090
```

Demo


index.html
Google Chrome
腾讯玄武实验室
TENCENT'S XUANWU LAB
在这里输入你要搜索的内容
英
16:49
2020/7/14

Agenda

- Undefined Behavior in LLVM
- Undefined Behavior Detections
- Detection Result Case Study
- From Undefined Behavior to RCE
- Conclusions

Suggestion

- For programmers:
  - Understand undefined behavior and write less UB bugs
- For compiler developers:
  - Provide more accurate and useful UB warnings to programmers
- For bug hunters:
  - Get more useful UB info from compilers

Summary

- Illustrate how to utilize compilers' capability to find UB bugs
- Explain several security bugs caused by UB
- Share advanced chromium exploitation techniques

Special Thanks

- Huiming Liu (@liuhm09)
- Chuanda Ding (@FlowerCode_)

Q & A

Thanks.

Tencent Security Xuanwu Lab
@XuanwuLab
xlab.tencent.com

Tencent 腾讯

腾讯安全玄武实验室
TENCENT SECURITY XUANWU LAB