Breached Labs

OSED Study Guide.

2025. A Breached Labs Resource.

What is the OSED

The Offensive Security Exploit Developer (OSED / EXP-301) certification focuses specifically on modern Windows x86 (32-bit) user-mode exploit development.

Unlike introductory exploit concepts that might only cover simple stack buffer overflows, OSED dives into techniques required to bypass common memory protections implemented in modern operating systems.

Key areas of focus include:

- **Bypassing Data Execution Prevention (DEP):** Techniques like Return-Oriented Programming (ROP) are central to OSED, allowing you to execute code even when the stack or other memory regions are marked as non-executable.
- **Bypassing Address Space Layout Randomization (ASLR):** Strategies to overcome the randomization of memory addresses for modules, the stack, and heap, often involving information leaks or targeting non-ASLR modules.
- **Structured Exception Handler (SEH) Overwrites:** Exploiting the Windows exception handling mechanism to gain control of execution flow.
- **Custom Shellcode:** Crafting position-independent shellcode that avoids bad characters and potentially uses techniques like egg hunters or API hashing.
- **Creative Problem Solving:** Utilizing techniques like finding code caves, using ROP-based decoders, and chaining various primitives together to achieve code execution under constraints.

OSED is less about *finding* the initial vulnerability (though some analysis is required) and more about the intricate process of turning a known vulnerability

(like a buffer overflow) into reliable code execution on a system with modern defenses enabled.

Why OSED

OSED forces you to grapple with low-level concepts in a way few other paths do. You'll gain an intimate understanding of process memory, assembly language, debuggers, and Windows internals.

Advanced exploit development, particularly for Windows environments and bypassing mitigations, is a specialized skill set highly valued in specific security roles. It differentiates you from those with more generalist cybersecurity knowledge.

Debugging complex exploits, chaining ROP gadgets, and overcoming unexpected hurdles requires meticulous attention to detail, persistence, and creative thinking. Skills which are applicable across many technical domains.

The OSED provides a strong foundation for tackling even more advanced topics like kernel exploitation, heap exploitation, or exploring different architectures (often covered in OSEE).

It's also helpful to understand where exactly the OSED fits relative to other popular Offensive Security certifications:

- **OSCP (Offensive Security Certified Professional):** Focuses on penetration testing methodologies. It covers network scanning, enumeration, exploiting various vulnerabilities across different systems (web, network services, Active Directory), and basic buffer overflows. OSCP is *broad*, testing the ability to compromise multiple machines in a

network. OSED is *deep*, focusing intensely on one specific area: advanced Windows x86 exploit development.

- **OSEP (Offensive Security Experienced Penetration Tester):** Focuses on advanced penetration testing techniques, including bypassing security solutions (like antivirus and EDR), advanced lateral movement, Active Directory attacks, and some client-side exploitation. While it involves exploiting systems, it doesn't delve into the ROP chains and mitigation bypasses to the depth that OSED does. OSEP is about sophisticated attack chains and evasion.
- **OSEE (Offensive Security Exploit Expert):** Considered the pinnacle of OffSec's exploit development track. It covers significantly more advanced topics, potentially including advanced heap exploitation, kernel-mode exploitation (Windows and/or Linux), patch diffing, and exploiting different architectures (e.g., ARM). OSED is often seen as a prerequisite or stepping stone towards the skills required for OSEE.

In essence: OSCP builds the pentesting foundation, OSEP advances adversary simulation skills, OSED dives deep into modern user-mode exploit development, and OSEE tackles the most advanced exploit challenges.

Career Paths

Holding an OSED certification demonstrates a high level of technical expertise and perseverance, opening doors to several specialized career paths:

- **Exploit Developer:** Designing and writing exploits for penetration testing teams, vulnerability research firms, or government/defense contractors.
- **Vulnerability Researcher:** Discovering new vulnerabilities (0-days) in software and systems. Understanding how to exploit vulnerabilities is often crucial for demonstrating their impact.

- **Security Researcher:** A broader role involving vulnerability research, mitigation techniques, tool development, and publishing findings. OSED skills are highly relevant.
- **Red Team Member (Specialist):** While general red teamers benefit, OSED holders can provide specialized capabilities, crafting custom exploits for engagements where standard tools fail, especially against hardened targets.
- **Reverse Engineer / Malware Analyst:** The deep understanding of low-level code execution and debugging gained through OSED is invaluable for analyzing complex software, including malware.

Ideal Path to OSED

While everyone's journey is unique, a common and effective path leading to OSED often looks like this:

1. **Foundational IT & Networking:** Basic understanding of operating systems, hardware, TCP/IP networking, client-server communication. You need to know how computers and networks fundamentally operate.
2. **Security Basics:** Familiarity with core security concepts (CIA triad, vulnerabilities, threats, risk), common attack vectors, and defensive measures.
3. **Penetration Testing (OSCP Level):** Practical experience in finding and exploiting vulnerabilities across various systems. This develops the "hacker mindset," enumeration skills, and crucially, often provides the first exposure to basic buffer overflows and the importance of persistence.
4. **Proficiency in C and Assembly (x86):** Dedicated study and practice with these low-level languages. This is where you learn how memory *really* works and how instructions translate to machine actions. *This*

step is absolutely critical and often runs parallel to or follows pentesting study.

5. **OSED (EXP-301):** With the above foundations, you are ready to tackle the specific challenges of modern Windows x86 exploit development, including DEP/ASLR bypasses and advanced shellcoding.

Skipping steps, particularly the low-level programming (C/Assembly) and foundational exploit concepts (basic buffer overflows), will make the OSED course material exponentially harder, potentially leading to frustration and failure. Solid foundational knowledge is paramount.

While OSCP/+ itself isn't a hard requirement if your other skills are exceptionally strong, the practical experience and basic exploit exposure it provides are highly beneficial.The OSCP does however provide exposure to basic buffer overflows (EIP overwrites), the OffSec methodology, practical problem-solving under pressure, and the crucial "Try Harder" mindset.

The Course

The core learning material typically consists of a detailed PDF document complemented by video demonstrations. The PDF serves as your primary textbook. It lays out the theoretical foundations, explains concepts in detail (e.g., ROP theory, SEH structure, egghunter logic), provides code examples, and walks through the initial steps of various exploitation techniques. Expect diagrams, assembly snippets, and explanations of Windows internals relevant to the exploits.

The videos demonstrate the practical application of the concepts explained in the PDF. Watching the instructors configure tools, debug applications, identify

issues, and build exploits step-by-step can significantly aid understanding. They often bridge the gap between theory and hands-on execution.
It's crucial to use both resources together. Read the relevant PDF section to grasp the concepts, then watch the corresponding video to see it in action.

The Lab

The OSED lab network is usually simpler than the sprawling OSCP labs. It focuses on providing specific target machines (Windows VMs) running vulnerable applications designed to teach the course concepts. You'll likely have a few different target VMs, each tailored for specific modules or techniques.

Types of Challenges:

- **Guided Examples:** These directly follow the course material (PDF/videos). Your goal is to replicate the steps shown, ensuring you understand each part of the process.
- **Extra Mile Exercises:** These often take the guided examples further, perhaps requiring you to adapt the exploit for a slightly different scenario, deal with new bad characters, or achieve a different objective. They test your understanding beyond simple replication.

It’s strongly recommended that you complete all guided examples, alongside the optional (but personally, mandatory) extra mile exercises.

You have **3 lab challenges** which you can complete. These will test you on the very topics which are covered by all the exam sets on which you can land on.

If you need help with any of the lab challenges, the OffSec Discord is a good place to ask for help. Explain clearly what you've tried and where you're

stuck. Avoid asking for direct answers; focus on understanding the concept you're missing.

Lastly: document *everything*. Your analysis steps, debugger commands, memory addresses, offsets, ROP gadget addresses, bad characters found, Python script versions, errors encountered, and how you fixed them. Use a structured note-taking app (CherryTree, OneNote, Obsidian).

The Exam

The OSED exam is a hands-on, practical test. You'll be given access to a separate exam VPN environment containing several vulnerable applications (typically 3 assignments, points vary).

You have a strict time limit (47 hours and 45 minutes) to develop working exploits. Following the exploitation phase, you have an additional 24 hours to write and submit a detailed professional report.

Typical exam challenges include:

- Handmade x86 Shellcode
- DEP/ASLR Buffer Overflow
- Reverse Engineering with IDA Pro

You need exactly 2 exploits, out of the 3 possible to pass the OSED exam.

Your report must include:

- Detailed steps for vulnerability analysis and exploit development.
- Explanation of the techniques used (ROP chain logic, shellcode function, etc.).

- Relevant code snippets (your exploit script, shellcode).
- Screenshots illustrating key steps (debugger state, successful execution like whoami or proof.txt).
- Clear, concise, professional language.

The Mindset

OSED is a marathon, not a sprint. It's challenging, and it's easy to get discouraged. Proactively manage your motivation:

- **Set Realistic Goals:** Break down the course into smaller, manageable chunks (e.g., "Master SEH overflows this week," "Complete the ROP module exercises").
- **Take Regular Breaks:** Step away from the keyboard, especially when frustrated. Go for a walk, do something unrelated. Often, solutions appear when you return with a fresh perspective.
- **Celebrate Small Wins:** Successfully replicating a guided exercise, finding your first ROP chain, getting shellcode to execute, acknowledge and appreciate these victories.
- **Connect with the Community:** Engage with fellow students on the OffSec forums or Discord. Sharing struggles and successes can be motivating (but avoid asking for spoilers!).
- **Focus on Learning, Not Just Passing:** Understand the underlying concepts deeply. The learning itself is valuable, regardless of the exam outcome on the first try.
- **Maintain Balance:** Don't let OSED consume your entire life. Maintain hobbies, relationships, and physical activity to avoid burnout.

Buffer Overflows

A rock-solid understanding of the stack overflow is the essential starting point. These concepts underpin almost everything that follows.

Identifying Stack Overflows:

- **Fuzzing Basics:** Sending increasingly long or malformed input to an application (especially network services or file parsers) to trigger crashes. Simple Python scripts using sockets or file manipulation can automate this. Look for crashes indicating potential control over the instruction pointer (EIP).
- **Crash Analysis:** When the application crashes under a debugger (like WinDbg or Immunity Debugger), examine the state of the registers, especially EIP and ESP, and the stack contents. If EIP contains parts of your input (e.g., 41414141 which is AAAA), you've likely found a stack overflow where you control the return address.

Understanding the Stack during Overflow:

When a function receives more data than its local buffer on the stack can hold, the excess data overwrites adjacent stack memory. This typically overwrites:

- Other local variables.
- The saved EBP (previous function's stack frame base pointer).
- The saved EIP (the return address – where execution should resume after the function finishes).
  Visualizing the stack layout (high addresses to low addresses: function arguments -> return address -> saved EBP -> local variables/buffer) is crucial.

Controlling Execution Flow:
The primary goal of a basic stack overflow is to overwrite the saved EIP on the stack with an address of your choosing. When the vulnerable function executes its ret instruction (which pops the return address off the stack into EIP), it will jump to *your* supplied address instead of returning normally. Usually, this address points to shellcode you've also injected into memory (often onto the stack itself, just after the EIP overwrite).

Dealing with Bad Characters:

They are characters that terminate or corrupt your payload when processed by the vulnerable application or protocol. Common examples include:

- **Null Byte (\x00):** Often terminates strings in C functions like strcpy. Almost always a bad character.
- **Line Feeds (\x0a), Carriage Returns (\x0d):** Can break network protocols or text input processing.
- **Whitespace (\x20, \x09):** Can be problematic depending on input parsing.
- **Protocol-Specific Characters:** Characters with special meaning in HTTP, FTP, etc. (e.g., /, ?, &).
- **Application-Specific Characters:** Some functions might filter or modify specific bytes (e.g., converting case).

Identifying Bad Characters Systematically:

Send a full byte range (\x01 through \xff) as part of your payload after achieving EIP control (pointing EIP to a buffer containing these bytes).

In the debugger, examine the memory where your byte range *should* be.

Identify any missing bytes or bytes that caused the sequence to truncate. The last working byte *before* the corruption or truncation indicates the bad character.

Remove the identified bad character and repeat the process until all bytes from \x01 to \xff arrive unmodified in memory. Remember \x00 is almost always bad.

Techniques for Handling/Avoiding:

Once bad characters are identified, you must ensure your EIP overwrite address, ROP gadget addresses, and shellcode do *not* contain them.

- **Shellcode Encoding:** Use encoders (like XOR, ADD/SUB) to transform shellcode into safe bytes, prepended with a small decoder stub that also avoids bad chars.
- **ROP Gadget Selection:** Choose ROP gadgets whose addresses do not contain bad characters. Tools like mona.py can help filter gadgets.
- **Alternate Instructions/Techniques:** Sometimes, you might need to find alternative ways to achieve a goal if the most direct method uses bad characters (e.g., using multiple ADD instructions instead of one MOV with a bad byte).

Static Analysis With IDA Pro

Familiarize yourself with the Graph view (visual control flow), Disassembly view (raw assembly), and Pseudocode view (decompiled C-like code - incredibly useful).

Look for known dangerous C functions (strcpy, sprintf, gets, memcpy with uncontrolled size, printf with format string issues, etc.). Use the tool's search and cross-reference features.

Trace program execution flow, identify how user input is processed, check boundary conditions, and understand data transformations (like custom encoding/decoding).

Locate buffer declarations (e.g., [ebp-100h]) in the disassembly or pseudocode near vulnerable functions to estimate required overflow sizes.

Extremely useful for seeing where a function is called *from* (xref from) or where a specific memory address/variable is accessed (xref to). Helps trace data flow and identify interesting code paths.

If the program uses custom encoding or checks, static analysis helps decipher the logic step-by-step.

Shellcode Development

Shellcode can't hardcode addresses due to ASLR and different Windows versions. **Techniques involve:**

- **Walking the PEB/TEB:** Use known offsets within the Process Environment Block (PEB) and Thread Environment Block (TEB) (accessed via the FS segment register) to find the base address of loaded modules like kernel32.dll.
- Parse the Export Address Table (EAT) of kernel32.dll to find LoadLibraryA and GetProcAddress.
- Use LoadLibraryA to load other needed DLLs (e.g., user32.dll).
- Use GetProcAddress to find the addresses of any other required API functions (e.g., WinExec, VirtualAlloc).

Hashing APIs: Instead of searching for function names (strings) in the EAT (which might contain bad characters or be suspicious), calculate a hash for each required function name. Your shellcode then walks the EAT, calculates the hash of each exported function name, and compares it to the target hash. This is more compact and stealthy.

Position-Independent Code (PIC):

- Use relative jumps (JMP short, JMP $+offset) and calls instead of absolute ones.
- Use CALL $+5 followed by POP ECX to get the current instruction pointer (EIP) into a register (ECX) for calculating addresses of strings or data embedded within the shellcode.
- Access embedded data using offsets from the obtained EIP (e.g., [ecx + offset_to_string]).

Debugging Shellcode:

- Use WinDbg or Immunity Debugger.
- Isolate the shellcode for testing if possible.
- Place an INT 3 (\xcc) breakpoint at the start.
- Step through instruction by instruction (t), monitoring registers (especially EAX for API return values) and memory to ensure it behaves as expected.
- Use NOP sleds (\x90) before shellcode during initial testing (if space permits) to slightly increase the chance of landing in executable code if your jump address is slightly off, but remove for final exploit unless needed.

SEH Exploitation

In a classic stack overflow scenario, if the overflow extends far enough up the stack, it can overwrite an SEH record. The goal is to:

1. Overwrite the SE Handler pointer with the address of code we control (e.g., a pointer to a POP POP RET gadget).

2. Overwrite the Next SEH Record pointer with a short JMP instruction that jumps over the overwritten SE Handler pointer to our shellcode placed immediately after it.

Finding POP POP RET Gadgets: When an exception is triggered and our overwritten SE Handler is called, the stack layout isn't immediately suitable for jumping directly to shellcode. A POP POP RET sequence is needed.

The two POP instructions remove the SE Handler and Next SEH pointers from the stack.

The RET instruction then pops the *next* value from the stack into EIP. If we place our short JMP instruction (which overwrote the original Next SEH record) correctly, this RET will jump to that JMP, which then jumps forward to our main shellcode.

Building SEH Exploits:

1. Identify a stack overflow vulnerability.
2. Determine the offset to overwrite the SEH record (Next SEH and SE Handler).
3. Find a suitable POP POP RET gadget address (no bad chars, preferably in a non-SafeSEH module if relevant).
4. Craft the payload: [JUNK] -> [Short JMP] (overwrites Next SEH) -> [Address of POP POP RET] (overwrites SE Handler) -> [Shellcode].
5. Trigger the exception (e.g., by causing an access violation after the overflow). The OS calls the overwritten handler (POP POP RET), which adjusts the stack and returns into the JMP, leading to the shellcode.

DEP Exploitation

Use ROP gadgets like POP EAX; RET, POP EBX; RET, etc. Place the desired value on the stack immediately after the gadget's address. When the gadget executes, it pops your value into the target register.

- **Basic ROP Chains:** The common goal is often to call VirtualProtect or VirtualAlloc from the Windows API:
- **VirtualProtect:** Change permissions on a memory region (e.g., the stack where your shellcode lies) to make it executable (RWX).
- **VirtualAlloc:** Allocate a *new* region of memory with executable permissions (RWX), copy shellcode there, and jump to it.
  To call an API function via ROP, you need gadgets to:
- Load the required arguments into the correct registers or onto the stack (following the function's calling convention).
- Load the address of the API function itself (e.g., VirtualProtect) into a register or onto the stack.
- Use a CALL EAX; RET or JMP ECX; RET style gadget, or simply RET directly to the API function address if placed correctly on the stack after arguments.

Building ROP Chains:

- Planning: Define your goal (e.g., call VirtualProtect). Identify the required arguments and their order. Find the address of the target API function (often needs to be fixed up if ASLR is also in play).
- Gadget Sequencing: Search for gadgets to load each argument. Sequence the gadget addresses and corresponding argument values meticulously on the stack. Ensure the stack pointer (ESP) is correctly managed by the gadgets.
- Stack Pivot Techniques: If your overflow space is limited or you need to place your ROP chain elsewhere (e.g., heap), use a

"stack pivot" gadget. Examples: XCHG EAX, ESP; RET (swaps EAX and ESP, useful if EAX points to your chain), MOV ESP, EBP; POP EBP; RET. This makes ESP point to your main ROP chain.

- Bad Characters: Crucially, *none* of the addresses used in your ROP chain (gadget addresses, API addresses) can contain bad characters identified earlier. This often heavily constrains gadget selection.

ASLR Exploitation

ASLR makes exploitation harder by randomizing the base addresses of key memory regions, making hardcoded addresses unreliable.**Use Non-ASLR Modules:** This is the most common and crucial technique for OSED. Check if the target application loads any DLLs that were compiled *without* the /DYNAMICBASE flag (meaning they are not ASLR-enabled).

These modules will load at the *same* predictable base address every time the application runs.

- **How:** Use tools like mona.py (!mona modules in Immunity/WinDbg) to list loaded modules and check their ASLR status.
- **Impact**: If a non-ASLR module exists, you can build your entire ROP chain using gadgets *only* from within that module, as their addresses will be static and reliable. You can also use addresses of functions exported by this module (like LoadLibraryA or GetProcAddress if available, though less likely).

Leaking Information: If no non-ASLR modules are available, you might need an *information leak* vulnerability. This is a separate flaw (e.g., format string vulnerability, buffer over-read sending back stack data) that discloses a

valid pointer from a randomized memory region (e.g., a stack address, or an address within a specific DLL like kernel32.dll).

- **How:** Exploit the info leak first to retrieve the pointer.
- **Impact:** Use the leaked address to calculate the actual randomized base address of the module it belongs to (leaked address - known offset = base address). Once you know the base address, you can calculate the runtime addresses of any needed ROP gadgets or API functions within that module for the current execution

---

Methodology

When faced with a new binary in the labs or exam, resist the urge to jump straight into exploit code. Follow a systematic approach:

1. **Reconnaissance:**
    - Understand the application's purpose and functionality. How is it meant to be used?
    - Identify input vectors: Network ports/protocols, file formats, command-line arguments, UI interactions. Where does user-controlled data enter the program?
    - Run the application normally, interact with it, observe its behavior. Use tools like netstat, Process Monitor if needed.
2. **Vulnerability Discover:**
    - **Fuzzing:** Send malformed or oversized data to the identified input vectors. Use simple scripts or basic fuzzing tools. Monitor for crashes.
    - **Static Analysis (IDA)**: Load the binary. Look for obvious vulnerable functions (strcpy, sprintf etc.). Analyze functions handling user input. Understand data flow from input to potential danger zones. Look for buffer size definitions.

- **Dynamic Analysis (WinDbg):** Attach the debugger. Set breakpoints before and during input processing. Step through code handling input to see how it's used.

3. **Crash Triage:**
    - Once a crash is triggered, reproduce it reliably under the debugger.
    - Analyze the crash state: Examine EIP, ESP, EBP, and other registers. Look at the stack (k, dd esp). Did EIP get overwritten with your input (e.g., 41414141)? Did an SEH record get overwritten?
4. **Developing the Exploit Primitive:**
    - Calculate the exact offset to control EIP or the SEH record using a cyclic pattern (msf-pattern_create/offset).
    - Confirm you can reliably control the instruction pointer (or the SE Handler pointer in SEH). Test by pointing it to a simple address like 0xDEADBEEF and verifying EIP contains that value upon crashing.
5. **Identifying and Overcoming Protections:**
    - Check loaded modules for DEP and ASLR status (!mona modules or similar). Are there any non-ASLR modules?
    - Determine the required bypass technique:
        - DEP Present -> ROP required.
        - ASLR Present & Non-ASLR module available -> Use gadgets from the non-ASLR module.
        - No DEP/ASLR (rare) -> Simple EIP overwrite to shellcode might work.
    - Identify bad characters specific to this vulnerability context.
6. **Shellcode Integration and Execution:**
    - Develop or generate shellcode, ensuring it avoids bad characters.
    - Encode shellcode if necessary and create a decoder stub (or plan a ROP decoder).

- Build the ROP chain (if needed) to bypass DEP (e.g., call VirtualProtect/VirtualAlloc), using only gadgets from non-ASLR modules if ASLR is enabled. Ensure all gadget addresses avoid bad characters.
- Integrate the components (Padding, EIP/SEH Overwrite, ROP Chain, Decoder, Shellcode, Egghunter if needed) into your exploit script.

7. **Refinement and Reliability:**
    - Test the exploit multiple times after reverting the target VM to ensure it works reliably.
    - Clean up exploit code, add comments.
    - Troubleshoot any lingering issues (timing, slightly wrong offsets, missed bad chars).

Additional Study Materials

Breached Labs

Corelan Team: (corelan.be): *Essential reading*. Their tutorials on stack overflows, SEH, ROP are legendary foundations.

FuzzySecurity: (fuzzysecurity.com/tutorials.html): Excellent Windows exploit dev tutorials.

Blogs: Search for exploit development writeups, specific technique explanations (e.g., egghunters, ROP techniques). Many security researchers share valuable insights. Plenty of writeups and exploit code can be found on Github.

VulnHub: Search for Windows machines tagged with "buffer overflow" or "exploit development".

Ending Notes

OSED is a significant milestone, but exploit development is a constantly evolving field. New mitigations arise, architectures change, and techniques adapt. Embrace the mindset of a lifelong learner. Stay curious, keep practicing, read research, and challenge yourself.

The path to OSED is challenging, demanding patience, meticulousness, and the famous "Try Harder" spirit. There will be moments of frustration, but the breakthroughs and the deep understanding gained are incredibly rewarding. Trust the process, rely on your fundamentals, practice diligently, and approach the challenges systematically.

Breached Labs