Noname manuscript No.
(will be inserted by the editor)

A Software Fault Tree Approach to Requirements Analysis of an Intrusion Detection System

Guy Helmer, Johnny Wong, Mark Slagell, Vasant Honavar, Les Miller, Robyn Lutz

e-mail: `{ghelmer, wong, slagell, honavar, lmiller, rlutz}@cs.iastate.edu`

The date of receipt and acceptance will be inserted by the editor

Abstract Requirements analysis for an Intrusion Detection System (IDS) involves deriving requirements for the IDS from analysis of the intrusion domain. When the IDS is, as here, a collection of mobile agents that detect, classify, and correlate system and network activities, the derived requirements include what activities the agent software should monitor, what intrusion characteristics the agents should correlate, where the IDS agents should be placed to feasibly detect the intrusions, and what countermeasures the software should initiate. This paper describes the use of software fault trees for requirements identification and analysis in an IDS. Intrusions are divided into seven stages (following Ruiu), and a fault subtree is developed to model each of the seven stages (reconnaissance, penetration, etc.). Two examples are provided. This approach was found to support requirements evolution (as new intrusions were identified), incremental development of the IDS, and prioritization of countermeasures.

Key words Software fault tree, requirements analysis, intrusion detection system, mobile agents, coloured petri nets

1 Introduction

A secure computer system provides guarantees regarding the confidentiality, integrity, and availability of its objects (such as data, processes, or services). However, systems generally contain design and implementation flaws that result in security vulnerabilities. An intrusion takes place when an attacker or group of attackers exploit security vulnerabilities and thus violate the confidentiality, integrity, or availability guarantees of a system. Intrusion detection systems (IDSs) detect some set of intrusions and execute some predetermined action when an intrusion is detected.

Intrusion detection systems use audit information obtained from host systems and networks to determine whether violations of a system's security policy are occurring or have occurred [1]. Our Multi-Agents Intrusion Detection System (MAIDS) [2] uses mobile agents [3] in a distributed system to obtain audit data, correlate events, and discover intrusions. The MAIDS system is comprised of (1) stationary data cleaning agents that obtain information from system logs, audit data, and operational statistics and convert the information into a common format, (2) low level agents that monitor and classify ongoing activities, classify events, and pass on their information to other agents, and (3) data mining agents that use machine learning to acquire predictive rules for intrusion detection from system logs and audit data.

However, we found the lack of a sound theoretical model and systematic method for the construction to be an impediment to development of the system in our early work. Existing intrusion detection systems tend to be built by selecting a set of data sources and developing a classification system to identify some set of intrusions using the selected data [2]. It is difficult to determine exactly what data elements should be correlated to determine whether an intrusion is taking place on a distributed system. It is also difficult to determine what data was necessary to discover intrusions. Verification of the proper operation of the IDS was possible only informally by executing the IDS and checking its results.

To bridge this gap, we started to look at IDS models. A model of intrusion detection is necessary to describe how the data should flow through the system, determine whether the system would be able to detect intrusions, and potentially suggest points at which countermeasures could be implemented. Such a model provides a formal specification of how to describe intrusions, identify intrusion characteristics and provably detect intrusions based on observable characteristics. Our approach begins with an analysis of intrusions to support a theoretical model of intrusion detection that answers questions about which intrusions are detectable, how they can be detected, how the data from different sensors should be correlated, and to what extent we can be assured that a report of an intrusion or a non-intrusion is accurate.

Software Fault Tree Analysis (SFTA) [4] is used in our approach to model intrusions and develop requirements for the IDS. SFTA is a method for identifying and documenting the combinations of lower-level software events that allow a top-level event (or root node) to occur. When the root node is a hazard, the SFTA assists in the requirements process by describing the known ways in which the system can reach that unsafe state. The safety requirements for the system can then be derived from the software fault tree, either indirectly [5,6] or directly via a shared model [7]. Software Fault Trees are closely related to threat trees [8].

In the work described here, we use SFTA to assist in determining and verifying the requirements for an intrusion detection system. The root node of the top-level SFTA is not strictly a hazard, as in a safety analysis, but an intrusion. An intrusion is a violation of a system's security policy. Intrusions result in compromise of exclusivity (unauthorized disclosure of data or use of services), integrity (unauthorized modification of data), or availability (denial of service). Whereas safety failures are often accidental or unexpected, intrusions are intentional, perpetrated by individuals, and can be expected to occur. Both safety and security failures represent potentially significant or catastrophic losses.

Intrusions can occur in a variety of ways. The software fault tree models the combinations and sequences of events by which intrusions can occur. The understanding and capture of domain knowledge needed to accurately define the requirements on an IDS is difficult. Questions such as what intrusions can be feasibly detected by the IDS software, at what stage of an intrusion the IDS software should detect each intrusion, and what assurances can be given that the IDS software detects intrusions must be addressed by the requirements analysis. The goal is not to build a system in which the root node never occurs, but to build an IDS in which the root node never occurs undetected.

The primary contribution of the work described here is to analyze the intrusion domain using software fault trees in order to determine the requirements for an IDS. The SFTA models the stages of intrusion in a structure that supports discovery and reasoning about requirements. In addition, the SFTA supports requirements evolution. The fault tree can be updated as new intrusions are identified, an essential feature for security applications. The SFTA also allows incremental development of the IDS as progressively more paths to the root node (or to the root node of a subtree) are blocked by the software. Inspection of the SFTA provides guidance as to where software monitors in the IDS should be required. Finally, path coverage metrics provide some verification that the IDS requirements are correct.

The rest of the paper is organized as follows. Section 2 provides some background on SFTA and graph-based IDS. Section 3 describes fault tree modeling of intrusions. Section 4 elaborates two specific examples from our experience with SFTA for the requirements determination of an IDS. Section 5 discusses the results of the use of SFTA in terms of requirements identification and analysis, requirements evolution, and verification. Section 6 contains concluding remarks.

2 Background

2.1 Software Fault Tree Analysis

The Software Fault Tree Analysis used to model intrusions is a backward search. It begins with a known hazard (here, an intrusion) as the root node and traces back through the possible parallel and serial combinations of events

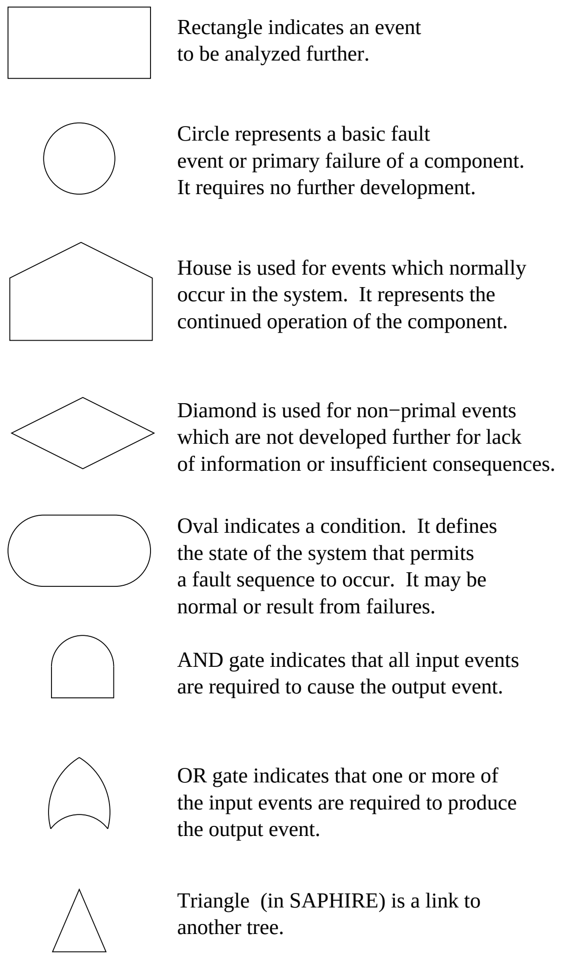

Rectangle indicates an event to be analyzed further.
Circle represents a basic fault event or primary failure of a component. It requires no further development.
House is used for events which normally occur in the system. It represents the continued operation of the component.
Diamond is used for non−primal events which are not developed further for lack of information or insufficient consequences.
Oval indicates a condition. It defines the state of the system that permits a fault sequence to occur. It may be normal or result from failures.
AND gate indicates that all input events are required to cause the output event.
OR gate indicates that one or more of the input events are required to produce the output event.
Triangle (in SAPHIRE) is a link to another tree.

Figure 1 Relevant fault tree symbols

that caused such an intrusion. The fault tree graphically represents this information in a diagram of events and logic gates leading to each hazard. Normally, the goals of developing a software fault tree include identifying contributing circumstances to an unsafe state and demonstrating that a system can not reach an unsafe state or that unsafe states are reached with very low probability [9]. In the intrusion domain, however, widely-deployed existing systems and protocols which are unsafe (i.e., allow intrusions) are modeled in the intrusion detection domain to enable reasoning about the possible combinations of events that lead to intrusions.

Figure 1 (a subset of available symbols [4]) shows commonly-used fault tree symbols. The procedure for fault tree analysis starts with identifying a hazard. The hazard becomes the root of the fault tree. Necessary preconditions for the hazard are specified in the next level of the tree and joined to the root with a logical AND or a logical OR. Each precondition is similarly expanded until all leaves are events that occur with some calculable probability or cannot be further analyzed. Fault tree analysis is used at the system level to identify high-level requirements for software safety. Software fault tree analysis is then performed on code, design, or requirements specifications [10]. The SAPHIRE software from the Idaho National Engineering and Environmental Lab was used to draw and edit the fault trees [11].

The OR gates in the fault trees shown are “true” if any input is true. The AND gates are “true” if all inputs are true in the current context, where the context may be a virtual network connection, a user’s login session, a series of related transactions, or some other temporal context. Child events of AND gates may take place in any order (left-to-right representation has been used where events occur in sequence, but the fault trees do not enforce the order). (Hansen et al. [7] discuss the ambiguities of traditionally accepted fault trees.)

A *cut set* is a set of basic events that causes the system to fail [12]. A cut set is called a *minimum cut set* if no basic event can be removed from the cut set and the root of the tree is still true [12]. A minimum cut of a fault tree gives a minimum set of successful events sufficient to satisfy the root. A minimum cut of an intrusion fault tree describes a scenario of a use case in which an attacker successfully exploits security flaws to achieve the goal of compromising the system. Manian, et al. [13] use Binary Decision Diagrams as an alternative to cutset-based solutions of fault trees for large, combinatorial solutions. However, in our current work, the size of the fault trees has been manageable using traditional cutset-based solutions.

2.2 Graph-Based Models of Intrusion Detection Systems

Several graph-based modeling techniques for IDS exist, but they model the intrusion detection system rather than the intrusion itself. For example, GrIDS, the Graph-based Intrusion Detection System, detects misuse in a system by dynamically building graphs that model the communication activities in a network [14]. The graph depends on user-defined rules to identify suspicious patterns and models intrusion detection, rather than intrusions. ARMD, the Adaptable Real-time Misuse Detection system, represents misuse signatures as directed acyclic graphs [15]. Unlike the object/event model used by GrIDS, the graphs are not amenable to aggregation. IDIOT, Intrusion Detection In Our Time, is an IDS that uses a custom language and a variant of Colored Petri Nets (CPNs) for misuse detection [16,17].

3 Developing Fault Trees for Intrusions

Intrusion fault tree modeling draws from a variety of sources. The standards used in current TCP/IP networks are publicly available. Proposals and standards for IP networks are published by the Internet Engineering Task Force (IETF) as Requests for Comment (RFC's) and Standards (STD's). Implementations of most network protocols are freely available in software such as Linux, FreeBSD, and Apache, allowing public review for security issues. Numerous researchers and hackers actively discover and publish security vulnerabilities in public forums including mail lists such as `bugtraq` and web sites such as `www.securityfocus.com`.

Faults that are generally UNIX-centric are considered in the fault trees, although many similar problems (e.g., buffer overflows) exist in software on other systems. Rather than looking directly at the source code for these systems, the immense body of publicly-discussed vulnerability information is used for development of the sample fault trees discussed here.

3.1 Reasonable Fault Trees

Each successful intrusion can vary greatly from all other intrusions, and attempts to analyze complete intrusions are difficult. A monolithic fault tree that would attempt to describe all intrusions would be huge, unwieldy, and less useful than several trees divided in a systematic manner. A reasonable approach is to divide intrusions into stages of intrusions that achieve intermediate goals of the attacker, and to develop fault trees that model each of the stages.

Ruiu's analysis of intrusions [18] separates intrusions into seven stages: (1) Reconnaissance, (2) Vulnerability Identification, (3) Penetration, (4) Control, (5) Embedding, (6) Data Extraction & Modification, and (7) Attack Relay. We use each of the seven stages as a root node in a SFTA, which has turned out to be a useful approach for the intrusions we have analyzed. Dividing stages in this manner seems to have been beneficial in that common subtrees can be identified and reused. Certain intrusion techniques (e.g., buffer overflow or printf-style format-string exploits) are often applied to many different components in a system.

3.2 SFTAs for Intrusions

The sample fault trees do not represent all possible combinations of events that make the root nodes true, even for known intrusions, but represent the known events in the documented intrusions against the systems of interest at this time. An example of how paths in the trees can describe a successful intrusion is discussed below.

3.2.1 Reconnaissance The reconnaissance phase identifies potential targets within an organization's networks. Network targets include not only multiuser hosts (e.g., UNIX or Windows/NT systems) but also routers, intelligent hubs, and perhaps even modems. The services offered by systems and names of users on the systems are also useful information for an attacker. Figure 2 shows a sample fault tree for the reconnaissance phase.

3.2.2 Vulnerability Identification Vulnerability identification is closely related to reconnaissance. In this phase, an attacker searches for vulnerabilities that can lead to penetration. The attacker sequentially scans many ports looking for versions of remote control services known to be vulnerable to intrusion (e.g., BackOffice or NetBus). Port scanning is a "noisy" monitorable intrusion, and is usually easy to detect unless done very slowly. The software fault tree in Figure 3 models this vulnerability identification phase of the intrusion (the FTPD node, number A5, is left unexpanded to allow us to show the diagram in a readable format). Other events, monitorable (such as checking of operating system version identification strings) and unmonitorable (such as off-site sniffing of network traffic) may also be used for version identification but were not included in Figure 3 for space reasons to make it readable.

3.2.3 Penetration Penetration occurs when an attacker obtains unauthorized access to a system. Penetration methods include exploitation of various network server daemon vulnerabilities (poor authentication and buffer overflows), authenticating with illicitly obtained passwords, and TCP session hijacking. Figures 4, 5, 6, and 7 together represent a sample fault tree for the penetration stage of intrusions. An example of potential subtree reuse can be seen in Figure 4, where node C2 (Shell Via Buffer Overflow) represents online access to a shell and also could be duplicated under node B2 (Execute Shell Code) where a shell could be used to execute a command.

3.2.4 Control An attacker needs to gain sufficient privilege in a system to continue to the next stages of the intrusion. Often an attacker must obtain privileges equivalent to those of the system administrator to gain sufficient control of a system. If the penetration was particularly effective and sufficient privilege was already gained, this step may not be necessary.

Mechanisms traced in Figure 8 include exploiting buffer overflows in privileged local programs, exploiting races in temporary files or signals, exploiting weak permissions on critical files and devices, and cracking a password for an administrator's account.

3.2.5 Embedding Embedding involves the installation or modification of a system so that even if the attacker is discovered and steps are taken to recover the system, the attacker will still be able to enter the system. For example, the system bootstrap code could be modified to re-insert backdoors if the system executable programs are restored from backups or installation media. Typical embedding techniques include installing Trojan horses, backdoor, and other rootkit programs, removing traces of the intrusion from system logs, and disabling detection systems.

Figure 9 identifies two different rootkit installations by matching particular sets of modified system files. A rootkit is a collection of embedding programs that allow an attacker to hide his activities and may include programs for use in the next step, data extraction & modification. If attackers do not typically install all the embedding programs in their kit, Figure 9 would need to be refined to reflect this case.

3.2.6 Data Extraction & Modification In the data extraction and modification phase, the attacker gathers information about the configuration and operation of the system. Covert channels may be used to move discovered data from the compromised system to the attacker's base. Events in this phase of an intrusion tend to resemble normal events (e.g., copying files). Anomaly detection seems to be an ideal application for detecting this stage of an intrusion. For these reasons, we concentrated on the other stages of intrusions and left this stage for later work.

3.2.7 Attack Relay After a system is fully compromised, it may be used for attack relaying. Intrusions can be launched against affiliated (trusting) hosts to expand the number of hosts under the attacker's control. A system also may simply be used to participate in distributed denial-of-service attacks [19,20,21,22]. Figure 10 represents some basic faults seen from Stacheldraht, Tribe Flood Network, and Trinoo distributed denial of service attacks. Many other forms of attack relaying exist, including automated and manual means.

4 Experience with Fault Trees for Intrusions

The relationship of the developed fault trees to the two intrusions is examined in this section. Each intrusion follows one of the multiple paths through each of the staged subtrees in Figures 2-10. A portion of the fault tree of Figure 7, describing the FTP bounce attack, was selected for further analysis. The FTP bounce attack subtree is particularly interesting because it involves several time-ordered steps which must take place for the intrusion to be successful.

4.1 Example 1: FTP SITE EXEC Intrusion

The FTP SITE EXEC attack against the wuftpd daemon is a buffer overflow attack [23]. When someone logs into the wuftpd daemon as the user `anonymous` or `ftp`, the daemon requests that the email address be entered as the password. However, an attacker can instead send malicious shell code in response to the password prompt. Then, if the SITE EXEC command is enabled, the attacker can send a SITE EXEC command with %-formatting characters that cause a buffer to overflow with data previously obtained as the password.

In the reconnaissance stage of the intrusion, an intruder discovers an anonymous FTP server host by using any one or more of the methods under the "HostDiscovery" node in the reconnaissance tree in Figure 2.

The intruder also discovers the availability of the FTP server by one of the methods under the "TCP-Service-Discovery" node in the reconnaissance tree.

In the next stage of the intrusion, the intruder identifies a vulnerability (a path through the subtree in Figure 3). The intruder may or may not take the time to make a connection to the FTP server and verify that the version number reported by the server is vulnerable to the FTP SITE EXEC attack. (Known FTP vulnerabilities are not expanded under the "FTPD" node in the vulnerability identification tree in Figure 3.)

Figures 4 to 7 show the penetration fault tree as this intrusion scenario continues (this scenario is shown in detail by a path in the fault tree of Figure 5). The intruder can now connect to the FTP server, give "anonymous" or "ftp" as the user name, and enter malicious shell code as the password. The intruder then issues a SITE EXEC command containing printf-style substitution character sequences. This is an attempt to overflow the character buffer on the process' stack with the data from the previously-entered "password." If the overflow is successful, the code provided by the intruder is executed with root privileges. A successful FTP SITE EXEC attack also gives the attacker control, so the attacker can move on to the later stages of the intrusion. (In this intrusion scenario, we assume the successful penetration results in privileged access, so the control phase of the intrusion may be by-passed.)

In the embedding stage, the intruder can install the Linux Rootkit version 4, which replaces a number of programs with Trojaned implementations that hide the attackers activities. Figure 9 shows the fault tree that matches the changes to the file system that result from the installation of the Linux Rootkit version 4.

In the data extraction stage, the intruder installs and runs a password sniffer that takes user names and passwords from telnet and ftp sessions on the LAN.

In the final stage of the intrusion, shown as a path through the intrusion fault tree in Figure 10, the intruder installs and runs a distributed denial of service agent, such as Trinoo, TFN, or Stacheldraht. The intruder can then use the system to execute attacks against other networked sites.

4.1.1 Derived IDS Requirements The software fault trees involved in this intrusion helped identify the software requirements for the mobile agent software tasked with detecting the FTP SITE EXEC intrusion. Examination of the penetration subtree shows that it is feasible to detect the FTP SITE EXEC attack in software.

4.1.2 Countermeasures for the FTP SITE EXEC intrusion Based on the derived requirements, an intrusion detection system should monitor PASS commands in an FTP session for data that does not represent a valid sequence of printable characters. That is, an invalid sequence of characters, including a number of characters beyond the maximum allowed password size or a password containing non-printable characters, is an event in any minimum cut set. The analysis does not say anything about how the monitoring should be implemented or performed; it merely leads to requirements for the intrusion detection system for stopping this particular intrusion.

4.2 Example 2: FTP Bounce Intrusion

The FTP bounce attack can be used to transfer data to a network port to which an attacker does not normally have access [24]. One way to exploit this problem is to send data to a remote shell server that trusts the FTP host via the FTP server. After an attacker discovers an FTP server and a host running rsh that might trust the FTP server, the attacker tries this exploit:

1. Uploads a specially-formatted file to the FTP server;
2. Issues an FTP PORT command that directs the FTP server to send its next download to port 514 on the target host;
3. Issues an FTP GET command to “download” the contents of the previously-uploaded file into port 514 on the target; the GET command opens a connection from the FTP server on port 20 to the rsh daemon on the target.
4. If the target trusts the FTP server, the rsh daemon will accept the contents of the file as if it were user input and execute the given command.

The following steps in an intrusion based on an FTP bounce attack show how the trees relate to the entire intrusion.

In the Reconnaissance stage of the intrusion, the intruder discovers an FTP server host and a target host, using any one or more of the methods under the “HostDiscovery” node in the reconnaissance tree. The intruder also discovers the availability of the FTP server and RSH server by one of the methods under the “TCP-Service-Discovery” node in the reconnaissance tree.

As in the intrusion discussed in Section 4.1, during the Vulnerability Identification stage, the intruder may or may not take the time to make a connection to the FTP server and verify that the version number reported by the server is vulnerable to the FTP bounce attack. In this path through the subtree, the intruder also needs a directory on the FTP server to which he or she may upload a file; if the intruder has no access to the FTP server other than “anonymous”, the intruder will have to search for such a directory.

The intruder will likely have to assume that the target host trusts the FTP server host, unless the intruder already has some access to the target host and can read the `/etc/hosts.equiv` or `~root/.rhosts` files. We have not considered “insider access” in the vulnerability identification tree.

The intrusion continues through the penetration subtree, with the intruder uploading the shell command file to the FTP server and issuing the appropriate FTP commands to cause the FTP server to “download” the file into the target’s RSH service. The “FTP-Bounce” subtree of Figure 7 shows the required FTP commands and responses. The structure of the subtree enforces the order of the events in the FTP command/response stream.

The successful FTP bounce intrusion mounted against a privileged account on the target also gives the attacker control, so the attacker can move on to the later stages of the intrusion. These later stages are omitted here for reasons of space since the path at this point is identical to that of the previously described intrusion.

4.2.1 Derived IDS Requirements Analysis of the subtree concerning the FTP bounce attack shows that, in order to detect the intrusion, the IDS needs to monitor commands and responses in an FTP session, to monitor rsh connections, and to correlate outputs from the two monitors to determine whether an FTP bounce attack was attempted and whether the intrusion was successful. As before, the analysis does not say anything about how the monitoring should be implemented or performed, but merely yields requirements for the intrusion detection system to stop or identify this particular intrusion.

4.2.2 Countermeasures for the FTP Bounce Attack Each of the steps in the intrusion detailed above is part of a scenario which fits a minimum cut of the corresponding fault tree. Inspecting the minimum cuts for each intrusion leads us to the best point at which to apply countermeasures. Countermeasures in intrusion detection systems typically include alerts to the system manager (via email, paging, or simply log messages), termination of network connections or logins, and disabling user accounts.

We examined the minimum cut from the penetration tree for the FTP Bounce Attack and informally considered the cost of applying countermeasures at each node. The cost included the complexity of the software required and the effect on the legitimate users of the system. It appears that the lowest cost countermeasure is to kill the TCP connection made from the FTP server to the RSH server; countermeasures at other nodes would either be prohibitive to implement, prevent legitimate uses of the FTP or RSH services, or be too late to terminate the FTP bounce intrusion.

5 Discussion of Results

Software fault trees for intrusions explore the sufficient combinations of events that lead to exploitation of a vulnerability. Development of fault trees for intrusions enabled a variety of discovery and verification activities. We summarize these briefly here and refer the reader to the previous section for examples.

5.1 Requirements Identification & Analysis

Fault trees document properties of intrusions and allow for analysis of intrusion properties.

Domain Understanding and Documentation.

Capturing this domain understanding is frequently difficult in the security arena. Software fault trees provide a standard, easy-to-use format for documenting properties of intrusions by system and network experts.

Determining Requirements.

Each minimum cut models an intrusion scenario that the software may be required to recognize. Identification of leaf events in the fault tree illustrates what components of a distributed system must be monitored to detect the intrusion. In addition, analysis of intrusion fault trees exposes conditions where countermeasures may be successfully applied by an intrusion detection system to intervene before the intrusion is successful.

Fault Detectability Analysis.

This refers to the ability of the system to detect the problem if it appears during system operation [25]. Determining which characteristics of intrusions can be monitored is an essential part of the requirements analysis for an IDS. For example, there exist certain intrusive events that do not have any discernible effect on a site's distributed system. Such events include "DNS zone transfers" from off-site secondary name servers and passive password sniffing. Marking these events appropriately in the fault tree allows analysis of which intrusions would be particularly difficult to detect, and may give hints regarding ways to prevent such intrusions from occurring.

5.2 Requirements Evolution & Incremental Development

Software fault trees support intrusion detection system development and maintenance activities.

Prioritization of Requirements.

The addition of historical likelihood and severity information [4] or risk vs. reward information [8] on the nodes (not addressed in this paper) assists in prioritizing requirements. In addition, based on this additional information from the fault trees, alert priorities may be encoded in an intrusion detection system. For example, most of the intrusions in the reconnaissance and vulnerability identification SFTAs have a low severity and high likelihood and are given a low alert priority. Conversely, the intrusions in the penetration and control SFTAs have a high severity and are given a high alert priority.

New intrusions.

Newly-discovered intrusions need to be integrated into the intrusion fault tree. Such new information may encourage re-organization of the fault tree, as when a new intrusion depends on a set of circumstances that is already diagrammed in the fault tree, or the addition of a subtree (either new or reused). The changes necessary in the intrusion fault tree to incorporate information about newly-discovered intrusions will then guide the necessary modifications of the intrusion detection requirements and design to detect the new intrusions.

5.3 Verification

Once confidence is established in a software fault tree, primarily through expert review, the design of the intrusion detection can then be traced to the software fault tree to determine its completeness and correctness.

Based on the testing strategy of Puketza et al. [26] the SFTA can be used to test the design and implementation of an IDS. Given a subtree of an SFTA that describes related intrusive events, define the subtree to be an equivalence class for the set of intrusions. Select one or more representative minimum cuts of the subtree to be tested. Then, given scenarios which are positive and negative examples of the intrusions, execute the intrusions and determine whether the subtree accurately matches the events. The scenarios form a set of representative test cases for the equivalence class.

We do not interpret the fault tree directly as requirements, unlike the approach used by Hansen et al. [7], where the fault tree has a formal semantics. A less formal approach was desired in the intrusion application because we want the fault tree to be developed and maintained by system support personnel rather than by experts in formal specification. It is primarily the support personnel's knowledge of the system and its vulnerabilities that the fault tree is intended to capture. To understand this, a brief description of the larger IDS system is in order.

The intrusion fault tree work described here is the requirements phase of a larger effort to provide a more formal framework for building IDS [27,28,29]. The IDS will use mobile agents in a distributed system to collect audit data, classify it, correlate information from the different mobile agents, and detect intrusions. The intrusion fault tree drives the requirements for these mobile agents and the intrusion detection system. The fault tree is mapped, by a correctness-preserving transformation, into Colored Petri Nets (CPNs) that serve as the design specification of the mobile agents in the IDS. Interactive simulation of these CPNs gives additional verification that the design satisfies the requirements (i.e., blocks the relevant path(s) in the intrusion fault tree). Code for the IDS mobile agents is generated from the CPNs and tested using, among other scenarios, the minimum cuts through the intrusion fault tree. Currently, prototypes exist of each of these phases (i.e., some CPNs and some mobile agents for some intrusions) with work on-going to partially automate the code generation.

6 Summary and Future Work

The use of software fault tree analysis to model intrusions to support requirements identification and analysis for an IDS has been presented with supporting examples and illustrative uses. Division of fault trees for intrusions into seven stages was examined, and sample fault trees for the intrusion stages were described. Using these staged subtrees, two intrusions were examined and software requirements for detection of the intrusions were derived from examination of the trees and associated minimum cut sets. An example use of SFTA for guiding countermeasures' requirements analysis was also described.

SFTAs enable structured analysis of intrusions and may be able to support both requirements evolution as new intrusions are added and to enable prioritized, incremental development of a distributed, agent-based IDS.

For our IDS prototypes, there has been no requirements specification. Instead, the intrusion fault trees have been interpreted as specifications of the combinations of events that must be detected. That is, the IDS requirements are that each of the intrusion sequences possible in the fault tree should be detected as soon (low in the tree) as possible. The leaf events describe what components of a distributed system must be monitored by the mobile agent software. No separate requirements specification document has been developed. Software fault tree models of intrusions provide an indirect requirements description for the design of the IDS. The resulting design is modeled on Colored Petri Nets and implementable in mobile agents. SFTA models of intrusions may also assist the verification process by providing test case scenarios (paths of intrusion) that the IDS is required to detect. Our ongoing research will determine the long-term effectiveness of SFTA in an IDS development cycle while automating the development process from requirements engineering through implementation. An example of the progress in this area is Slagell's technical report [28].

We have begun to formalize the use of the developed software fault trees to drive the development of an intrusion detection design. We are examining extending SFTAs with additional information. Without this additional system-specific information, the IDS yields many false positives, detecting intrusions where, in a specific network, there is none. More information on these constraints is available in Helmer's dissertation [27].

The SFTAs developed in this work deal with known, staged misuse intrusions. Ideas for future research in the same view include developing models of anomaly intrusions and non-staged misuse intrusions. Non-staged misuse intrusions include intrusions such as denials of service. Such intrusions do not follow the seven stages of an intrusion but violate the availability, confidentiality, or integrity of a computing system. Some of these intrusions have been exceedingly effective but difficult to detect and counter. Developing a model of these intrusions may assist the development of detection and countermeasures.

Detecting as-yet-unknown misuse intrusions may be assisted by SFTA. As an expert constructs a fault tree, he or she should consider reasonable (but as yet unnoticed) events that could contribute to an intrusion. An interesting possibility for further research would be to build an SFTA in this way and evaluate it against existing systems to determine whether hypothesized vulnerabilities do, in fact, exist.

A related aspect of SFTA development is the tedious, detailed work and expert analysis required. We are interested in researching machine learning approaches to support automated development of SFTA.

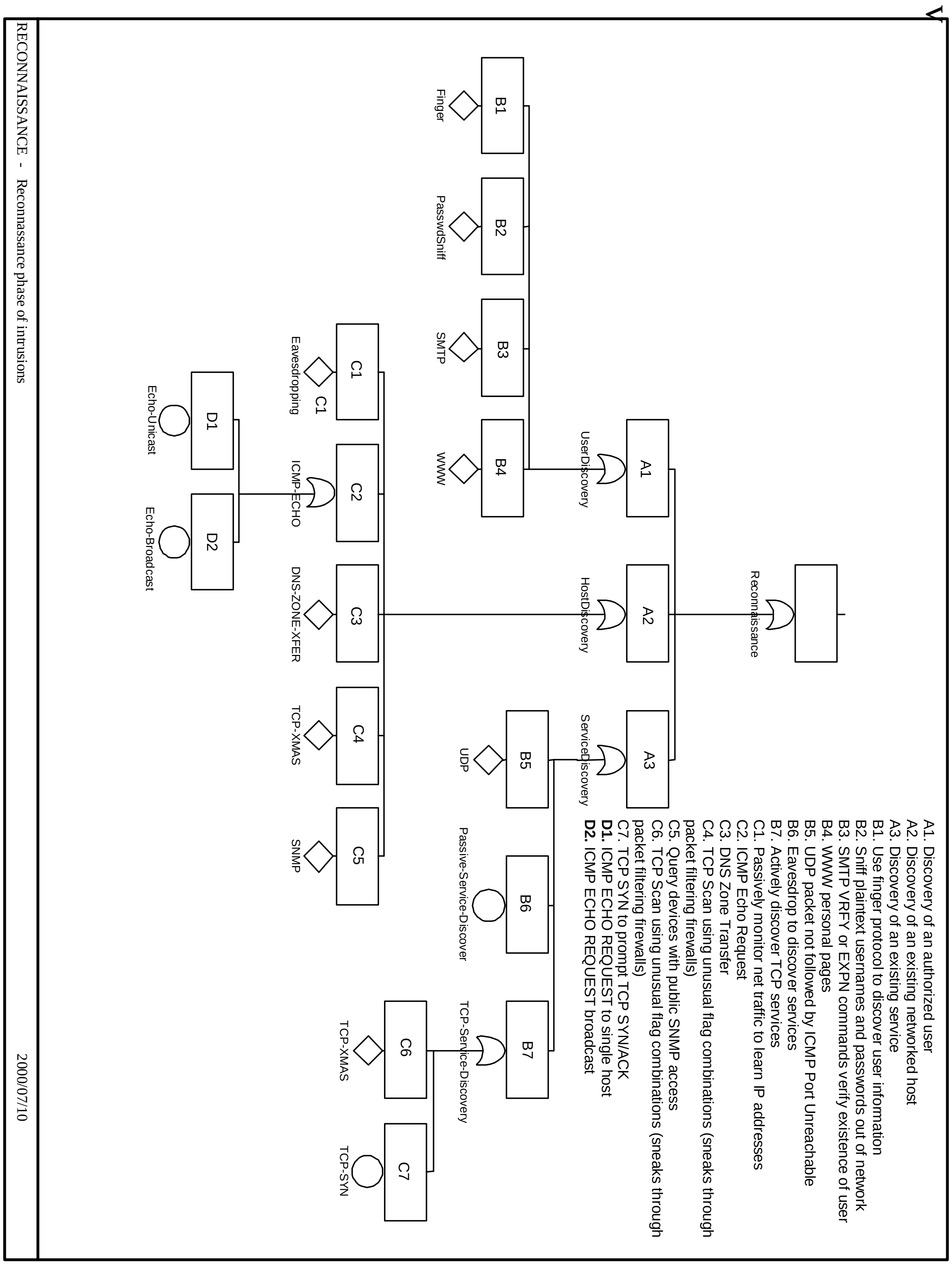
Anomaly intrusion detection systems are a subject of current research activity, but as with misuse IDSs, tend to start with a particular data source and match an intrusion detection approach to the data. Analysis and modeling of anomaly intrusions may assist and improve the development of anomaly IDSs.

Acknowledgements Funded in part by the Department of Defense. This work was performed in part by the Jet Propulsion Laboratory, California Institute of Technology, under a contract with NASA. Funded in part by NASA's Code Q Software Program Center Initiative, UPN 323-08. The assistance of Palisade Systems, Inc. is gratefully acknowledged. An earlier version of the paper was presented at the 1st Symposium on Requirements Engineering for Information Security, 2001, Indianapolis, Indiana USA.

References

1. Amoroso, E. Intrusion Detection. Intrusion.Net Books, Sparta, NJ, USA, 1999.
2. Helmer, G., Wong, J. S. K., Honavar, V., and Miller, L. Intelligent agents for intrusion detection. In: Proceedings, IEEE Information Technology Conference, Syracuse, NY, USA, Sept. 1998, pp. 121–124.
3. Bradshaw, J. M., Ed. An Introduction to Software Agents. MIT Press, Cambridge, MA, USA, 1997.
4. Leveson, N. G. Safeware: System Safety and Computers. Addison-Wesley, Reading, MA, USA, 1995.
5. De Lemos, R., Saeed, A., and Anderson, T. Analyzing safety requirements for process-control systems. IEEE Software 1995; 12(3):42–53.
6. Lutz, R., and Woodhouse, R. M. Requirements analysis using forward and backward search. Annals of Software Engineering 1997; 3:459–475.
7. Hansen, K. M., Ravn, A. P., and Stavridou, V. From safety analysis to software requirements. IEEE Transactions on Software Engineering July 1998; 24(7):573–584.
8. Amoroso, E. Fundamentals of Computer Security Technology. Prentice-Hall PTR, Upper Saddle River, NJ, 1994.
9. Leveson, N. G., Cha, S. S., and Shimeall, T. J. Safety verification of Ada programs using software fault trees. IEEE Software July 1991; 8(4):48–59.
10. Lutz, R. R. Targeting safety-related errors during software requirements analysis. Journal of Systems and Software Sept. 1996; 34:223–230.
11. Idaho National Engineering and Environmental Laboratory. SAPHIRE - systems analysis programs for hands-on integrated reliability evaluations. Online, 2000. `http://saphire.inel.gov/`.
12. Raheja, D. G. Assurance Technologies: Principles and Practices. McGraw-Hill Engineering and Technology Management Series. McGraw-Hill, New York, 1991.

13. Manian, R., Dugan, J. B., Coppit, D., and Sullivan, K. J. Combining various solution techniques for dynamic fault tree analysis of computer systems. In: 3rd IEEE International High-Assurance Systems Engineering Symposium, IEEE Computer Society, 1998, pp. 21–28.
14. Staniford-Chen, S., Cheung, S., Crawford, R., et al. GrIDS-a graph based intrusion detection system for large networks. In: 19th National Information Systems Security Conference Proceedings, Oct. 1996, pp. 361–370.
15. Lin, J.-L., Wang, X. S., and Jajodia, S. Abstraction-based misuse detection: High-level specifications and adaptable strategies. In: Proceedings, IEEE Computer Security Foundations Workshop, Rockport, MA, USA, June 1998, pp. 190–201.
16. Kumar, S., and Spafford, E. H. A pattern matching model for misuse intrusion detection. In: Proceedings of the 17th National Computer Security Conference, Baltimore, MD, USA, Oct. 1994, pp. 11–21.
17. Kumar, S. Classification and Detection of Computer Intrusions. PhD thesis, Purdue University, West Lafayette, IN, USA, Aug. 1995.
18. Ruiu, D. Cautionary tales: Stealth coordinated attack howto, July 1999. `http://www.nswc.navy.mil/ISSEC/CID/Stealth_Coordinated_Attack.html`.
19. Dittrich, D., Weaver, G., Dietrich, S., and Long, N. The “mstream” distributed denial of service attack tool. Online, May 2000. `http://staff.washington.edu/dittrich/misc/mstream.analysis.txt`.
20. Dittrich, D. The “stacheldraht” distributed denial of service attack tool. Online, Dec. 1999. `http://staff.washington.edu/dittrich/misc/stacheldraht.analysis.txt`.
21. Dittrich, D. The “tribe flood network” distributed denial of service attack tool. Online, Oct. 1999. `http://staff.washington.edu/dittrich/misc/tfn.analysis.txt`.
22. Dittrich, D. The DoS Project’s “trinoo” distributed denial of service attack tool. Online, Oct. 1999. `http://staff.washington.edu/dittrich/misc/trinoo.analysis.txt`.
23. CERT Coordination Center. Two input validation problems in FTPD. Online, July 2000. `http://www.cert.org/advisories/CA-2000-13.html`.
24. CERT Coordination Center. FTP bounce. Online, Dec. 1997. `http://www.cert.org/advisories/CA-97.27.FTP_bounce.html`.
25. Del Gobbo, D., Cukic, B., Napolitano, M. R., and Easterbrook, S. Fault detectability analysis for requirements validation of fault tolerant systems. In: 4th IEEE International High-Assurance Systems Engineering Symposium, IEEE Computer Society, 1999, pp. 231–238.
26. Puketza, N. J., Zhang, K., Chung, M., Mukherjee, B., and Olsson, R. A. A methodology for testing intrusion detection systems. IEEE Transactions on Software Engineering Oct. 1996; 22(10):719–729.
27. Helmer, G. Intelligent multi-agent system for intrusion detection and countermeasures. PhD thesis, Iowa State University, Ames, IA, USA, Dec. 2000.
28. Slagell, M. The design and implementation of MAIDS (mobile agent intrusion detection system). Tech. Rep. TR01-07, Iowa State University Department of Computer Science, Ames, IA, USA, 2001.
29. Helmer, G., Wong, J., Honavar, V., and Miller, L. Lightweight agents for intrusion detection. To appear, Journal of Systems and Software, 2003.



A1. Discovery of an authorized user
A2. Discovery of an existing networked host
A3. Discovery of an existing service
B1. Use finger protocol to discover user information
B2. Sniff plaintext usernames and passwords out of network
B3. SMTP VRFY or EXPN commands verify existence of user
B4. WWW personal pages
B5. UDP packet not followed by ICMP Port Unreachable
B6. Eavesdrop to discover services
B7. Actively discover TCP services
C1. Passively monitor net traffic to learn IP addresses
C2. ICMP Echo Request
C3. DNS Zone Transfer
C4. TCP Scan using unusual flag combinations (sneaks through packet filtering firewalls)
C5. Query devices with public SNMP access
C6. TCP Scan using unusual flag combinations (sneaks through packet filtering firewalls)
C7. TCP SYN to prompt TCP SYN/ACK
D1. ICMP ECHO REQUEST to single host
D2. ICMP ECHO REQUEST broadcast

Figure 2 Reconnaissance fault tree

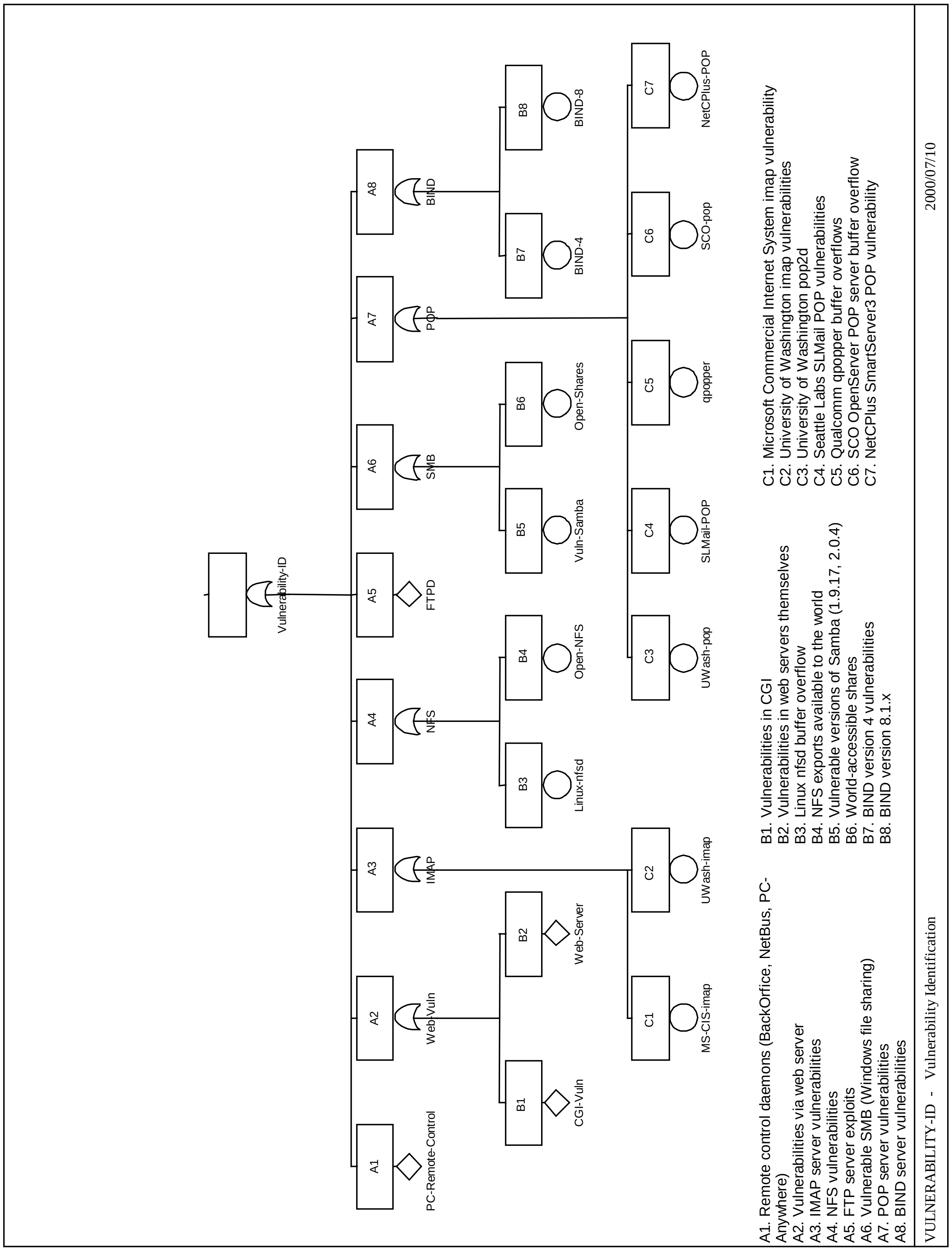

Vulnerability-ID
A1
PC-Remote-Control
A2
Web-Vuln
A3
IMAP
A4
NFS
A5
FTPD
A6
SMB
A7
POP
A8
BIND
B1
CGI-Vuln
B2
Web-Server
B3
Linux-nfsd
B4
Open-NFS
B5
Vuln-Samba
B6
Open-Shares
B7
BIND-4
B8
BIND-8
C1
MS-CIS-imap
C2
UWash-imap
C3
UWash-pop
C4
SLMail-POP
C5
qpopper
C6
SCO-pop
C7
NetCPlus-POP
A1. Remote control daemons (BackOffice, NetBus, PC-Anywhere)
A2. Vulnerabilities via web server
A3. IMAP server vulnerabilities
A4. NFS vulnerabilities
A5. FTP server exploits
A6. Vulnerable SMB (Windows file sharing)
A7. POP server vulnerabilities
A8. BIND server vulnerabilities
B1. Vulnerabilities in CGI
B2. Vulnerabilities in web servers themselves
B3. Linux nfsd buffer overflow
B4. NFS exports available to the world
B5. Vulnerable versions of Samba (1.9.17, 2.0.4)
B6. World-accessible shares
B7. BIND version 4 vulnerabilities
B8. BIND version 8.1.x
C1. Microsoft Commercial Internet System imap vulnerability
C2. University of Washington imap vulnerabilities
C3. University of Washington pop2d
C4. Seattle Labs SLMail POP vulnerabilities
C5. Qualcomm qpopper buffer overflows
C6. SCO OpenServer POP server buffer overflow
C7. NetCPlus SmartServer3 POP vulnerability
VULNERABILITY-ID - Vulnerability Identification
2000/07/10

Figure 3 Vulnerability identification fault tree

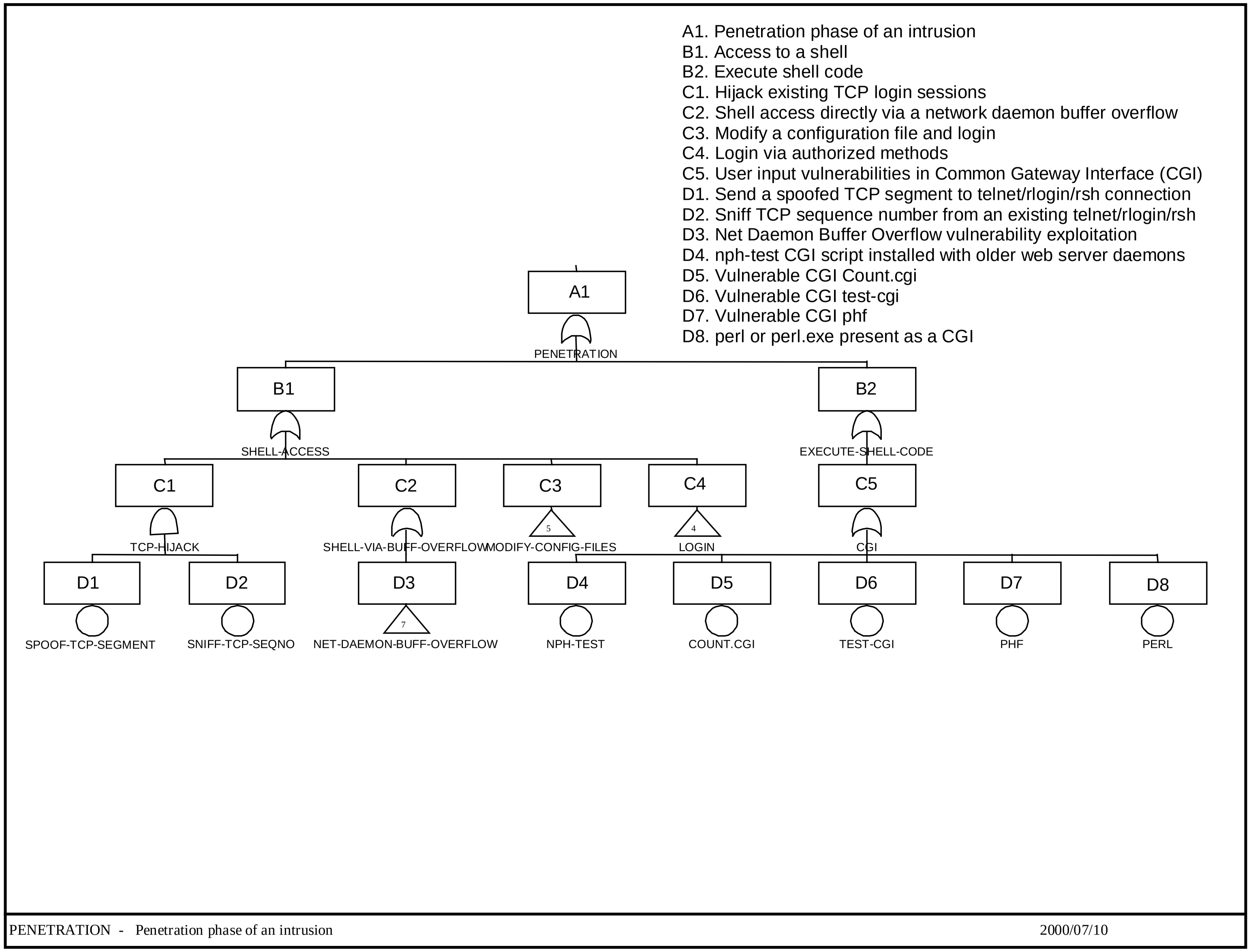

A1. Penetration phase of an intrusion
B1. Access to a shell
B2. Execute shell code
C1. Hijack existing TCP login sessions
C2. Shell access directly via a network daemon buffer overflow
C3. Modify a configuration file and login
C4. Login via authorized methods
C5. User input vulnerabilities in Common Gateway Interface (CGI)
D1. Send a spoofed TCP segment to telnet/rlogin/rsh connection
D2. Sniff TCP sequence number from an existing telnet/rlogin/rsh
D3. Net Daemon Buffer Overflow vulnerability exploitation
D4. nph-test CGI script installed with older web server daemons
D5. Vulnerable CGI Count.cgi
D6. Vulnerable CGI test-cgi
D7. Vulnerable CGI phf
D8. perl or perl.exe present as a CGI
A1
PENETRATION
B1
B2
SHELL-ACCESS
EXECUTE-SHELL-CODE
C1
C2
C3
C4
C5
5
4
TCP-HIJACK
SHELL-VIA-BUFF-OVERFLOW
MODIFY-CONFIG-FILES
LOGIN
CGI
D1
D2
D3
D4
D5
D6
D7
D8
7
SPOOF-TCP-SEGMENT
SNIFF-TCP-SEQNO
NET-DAEMON-BUFF-OVERFLOW
NPH-TEST
COUNT.CGI
TEST-CGI
PHF
PERL
PENETRATION - Penetration phase of an intrusion
2000/07/10

Figure 4 Penetration fault tree

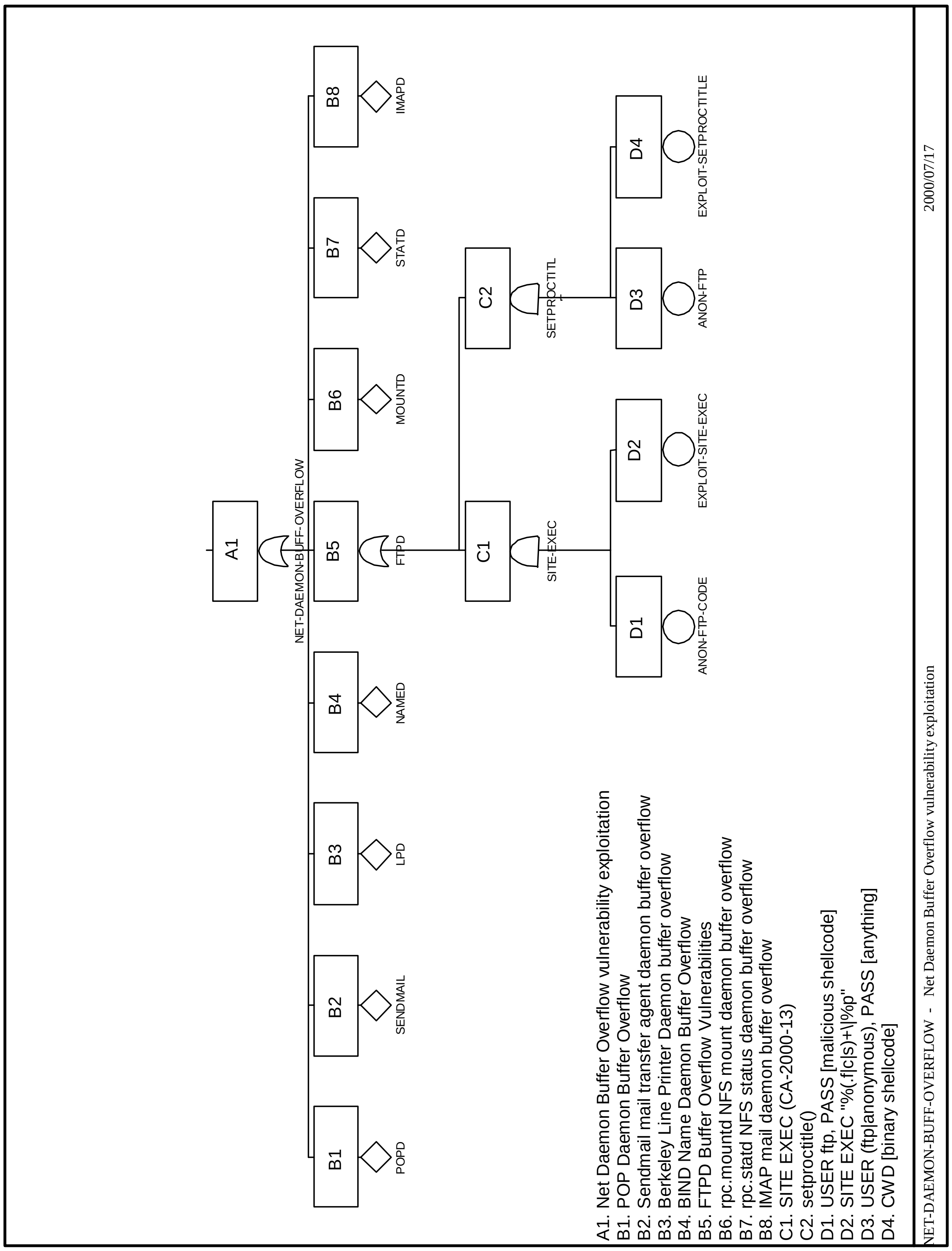
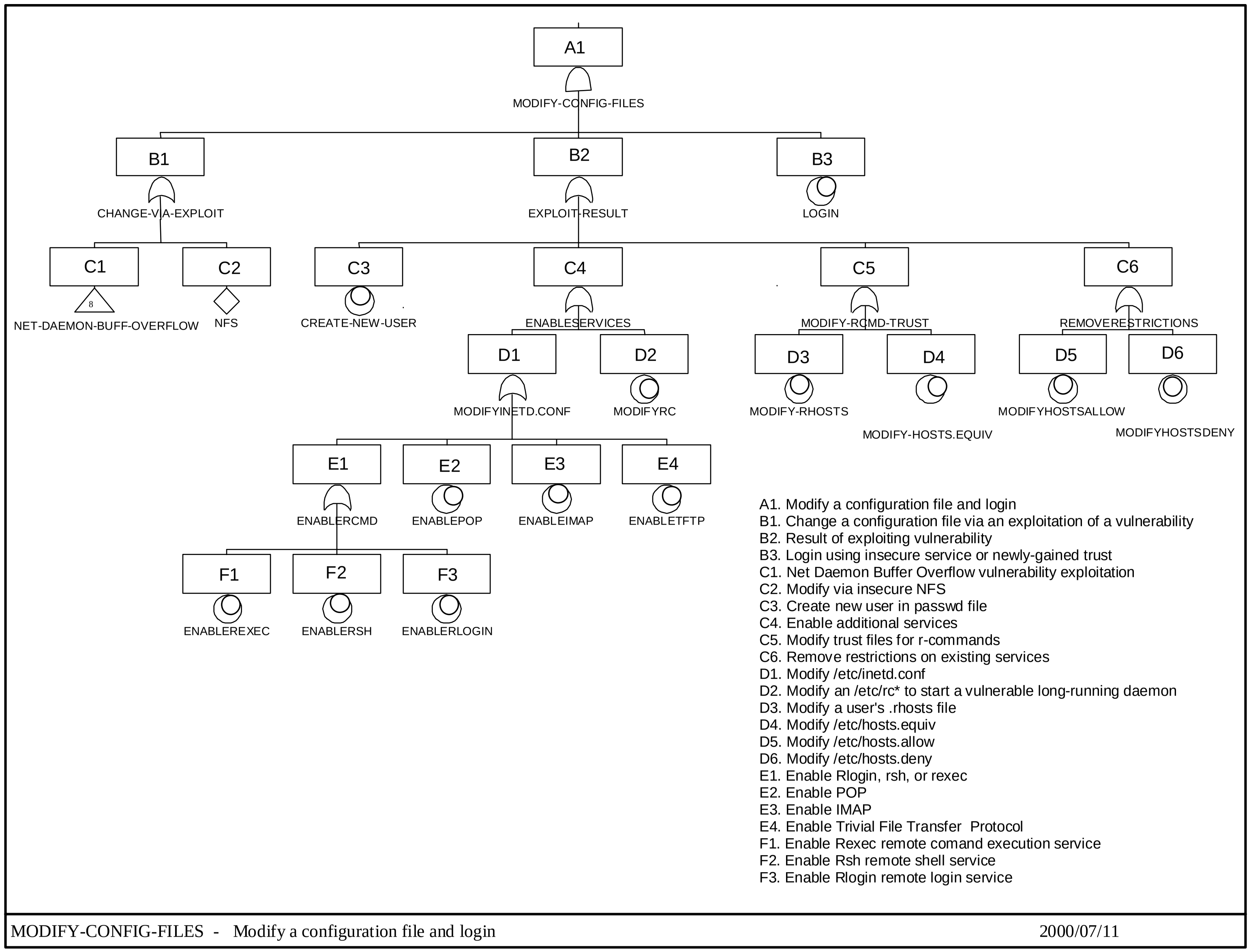
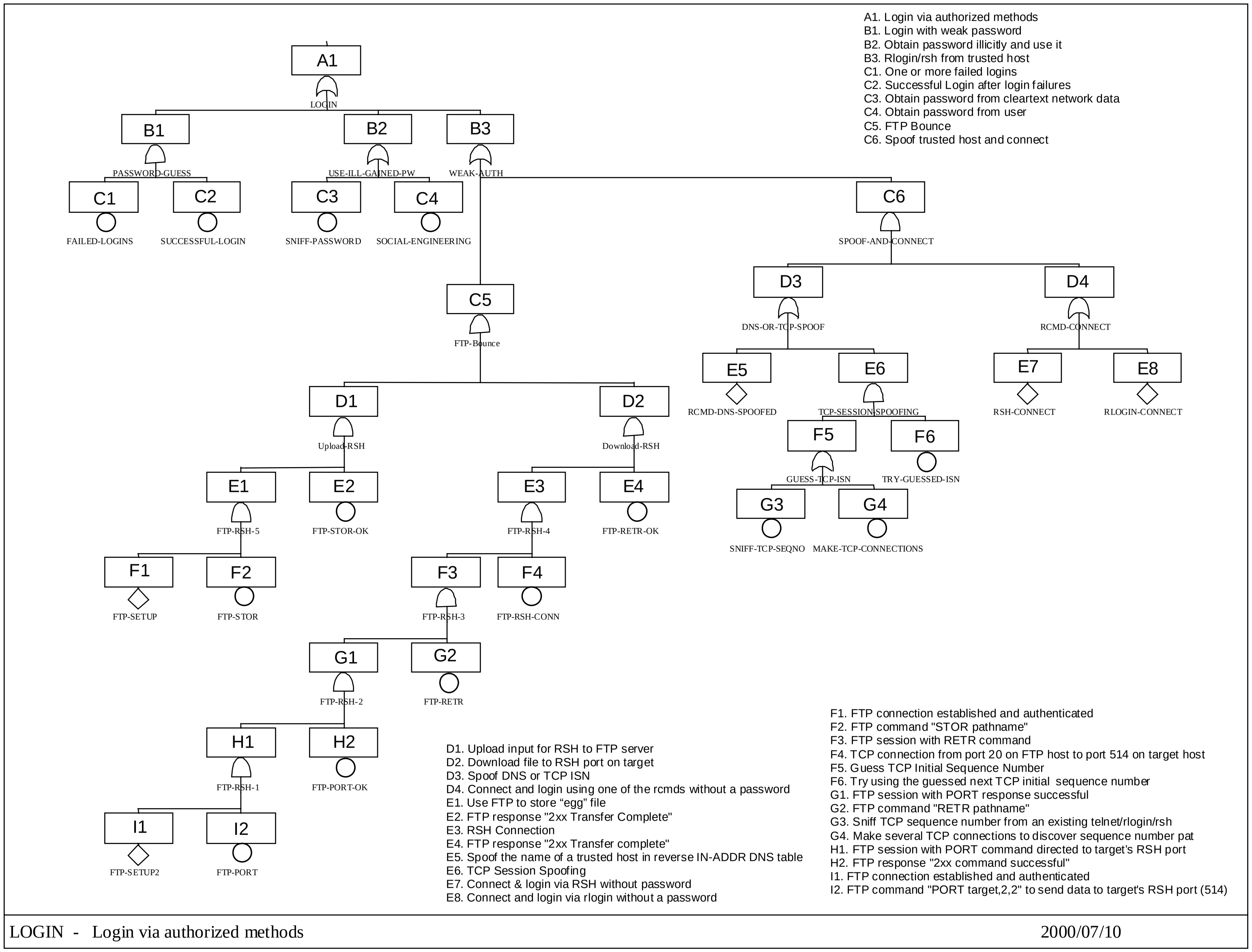

A1
NET-DAEMON-BUFF-OVERFLOW
B1
POPD
B2
SENDMAIL
B3
LPD
B4
NAMED
B5
FTPD
B6
MOUNTD
B7
STATD
B8
IMAPD
C1
SITE-EXEC
C2
SETPROCTITLE
D1
ANON-FTP-CODE
D2
EXPLOIT-SITE-EXEC
D3
ANON-FTP
D4
EXPLOIT-SETPROCTITLE
A1. Net Daemon Buffer Overflow vulnerability exploitation
B1. POP Daemon Buffer Overflow
B2. Sendmail mail transfer agent daemon buffer overflow
B3. Berkeley Line Printer Daemon buffer overflow
B4. BIND Name Daemon Buffer Overflow
B5. FTPD Buffer Overflow Vulnerabilities
B6. rpc.mountd NFS mount daemon buffer overflow
B7. rpc.statd NFS status daemon buffer overflow
B8. IMAP mail daemon buffer overflow
C1. SITE EXEC (CA-2000-13)
C2. setproctitle()
D1. USER ftp, PASS [malicious shellcode]
D2. SITE EXEC "%(.f|c|s)+\|%p"
D3. USER (ftp|anonymous), PASS [anything]
D4. CWD [binary shellcode]
NET-DAEMON-BUFF-OVERFLOW - Net Daemon Buffer Overflow vulnerability exploitation
2000/07/17

Figure 5 Penetration fault tree: Using buffer overflows in network daemons



A1. Modify a configuration file and login
B1. Change a configuration file via an exploitation of a vulnerability
B2. Result of exploiting vulnerability
B3. Login using insecure service or newly-gained trust
C1. Net Daemon Buffer Overflow vulnerability exploitation
C2. Modify via insecure NFS
C3. Create new user in passwd file
C4. Enable additional services
C5. Modify trust files for r-commands
C6. Remove restrictions on existing services
D1. Modify /etc/inetd.conf
D2. Modify an /etc/rc* to start a vulnerable long-running daemon
D3. Modify a user's .rhosts file
D4. Modify /etc/hosts.equiv
D5. Modify /etc/hosts.allow
D6. Modify /etc/hosts.deny
E1. Enable Rlogin, rsh, or rexec
E2. Enable POP
E3. Enable IMAP
E4. Enable Trivial File Transfer Protocol
F1. Enable Rexec remote comand execution service
F2. Enable Rsh remote shell service
F3. Enable Rlogin remote login service

Figure 6 Penetration fault tree: Gaining access by modifying configuration files



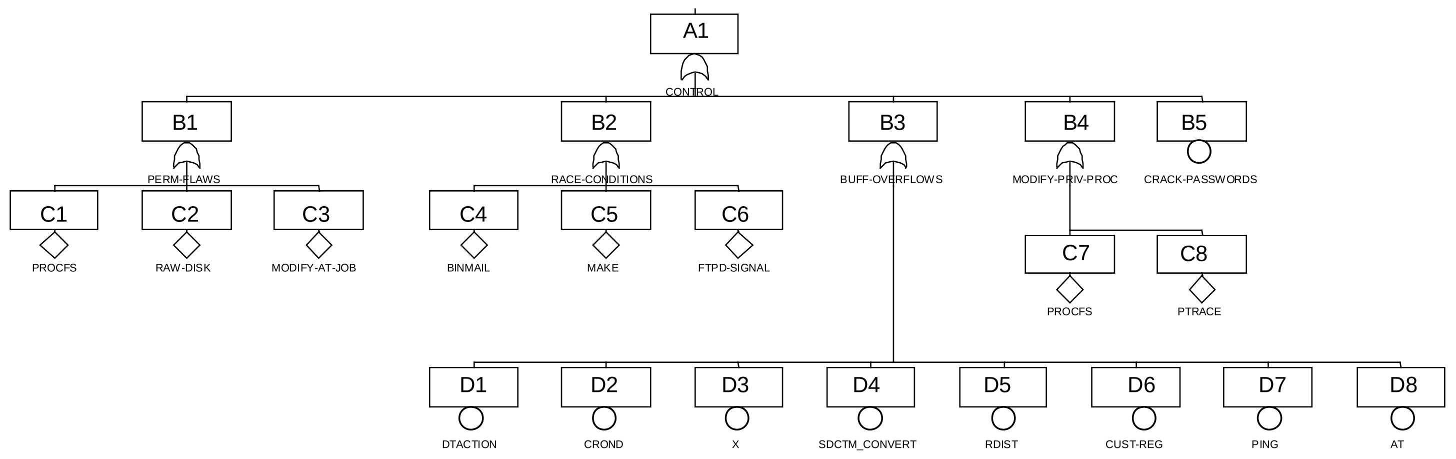
A1. Login via authorized methods
B1. Login with weak password
B2. Obtain password illicitly and use it
B3. Rlogin/rsh from trusted host
C1. One or more failed logins
C2. Successful Login after login failures
C3. Obtain password from cleartext network data
C4. Obtain password from user
C5. FTP Bounce
C6. Spoof trusted host and connect

D1. Upload input for RSH to FTP server
D2. Download file to RSH port on target
D3. Spoof DNS or TCP ISN
D4. Connect and login using one of the rcmds without a password
E1. Use FTP to store “egg” file
E2. FTP response "2xx Transfer Complete"
E3. RSH Connection
E4. FTP response "2xx Transfer complete"
E5. Spoof the name of a trusted host in reverse IN-ADDR DNS table
E6. TCP Session Spoofing
E7. Connect & login via RSH without password
E8. Connect and login via rlogin without a password

F1. FTP connection established and authenticated
F2. FTP command "STOR pathname"
F3. FTP session with RETR command
F4. TCP connection from port 20 on FTP host to port 514 on target host
F5. Guess TCP Initial Sequence Number
F6. Try using the guessed next TCP initial sequence number
G1. FTP session with PORT response successful
G2. FTP command "RETR pathname"
G3. Sniff TCP sequence number from an existing telnet/rlogin/rsh
G4. Make several TCP connections to discover sequence number pat
H1. FTP session with PORT command directed to target's RSH port
H2. FTP response "2xx command successful"
I1. FTP connection established and authenticated
I2. FTP command "PORT target,2,2" to send data to target's RSH port (514)

LOGIN - Login via authorized methods 2000/07/10

Figure 7 Penetration fault tree: Gaining access through abuse of authentication methods



A1. Gain system privileges
B1. Exploit flaws in file, directory, or registry permissions
B2. Exploit race conditions in privileged processes
B3. Exploit buffer overflows in privileged (setuid) programs
B4. Modify a privileged process during execution
B5. Run password cracker on local passwords
C1. Modify privileged process via procfs
C2. Read/write raw disk (NetBSD)
C3. Modify jobs in /var/at/spool (Linux, FreeBSD, NetBSD)
C4. /bin/mail symlink vulnerability
C5. make(1) temporary files
C6. Signals to ftpd allow reading & writing files as root
C7. Modify privileged process via procfs
C8. Modify privileged process via ptrace(2)
D1. Solaris dtaction
D2. Linux crond
D3. X server
D4. Solaris scdtm_convert
D5. Solaris rdist
D6. SGI On-Line Customer Registration program
D7. Solaris ping
D8. at(1) on AIX, SCO, Solaris

CONTROL - Control phase of intrusions 2000/07/10

Figure 8 Control fault tree

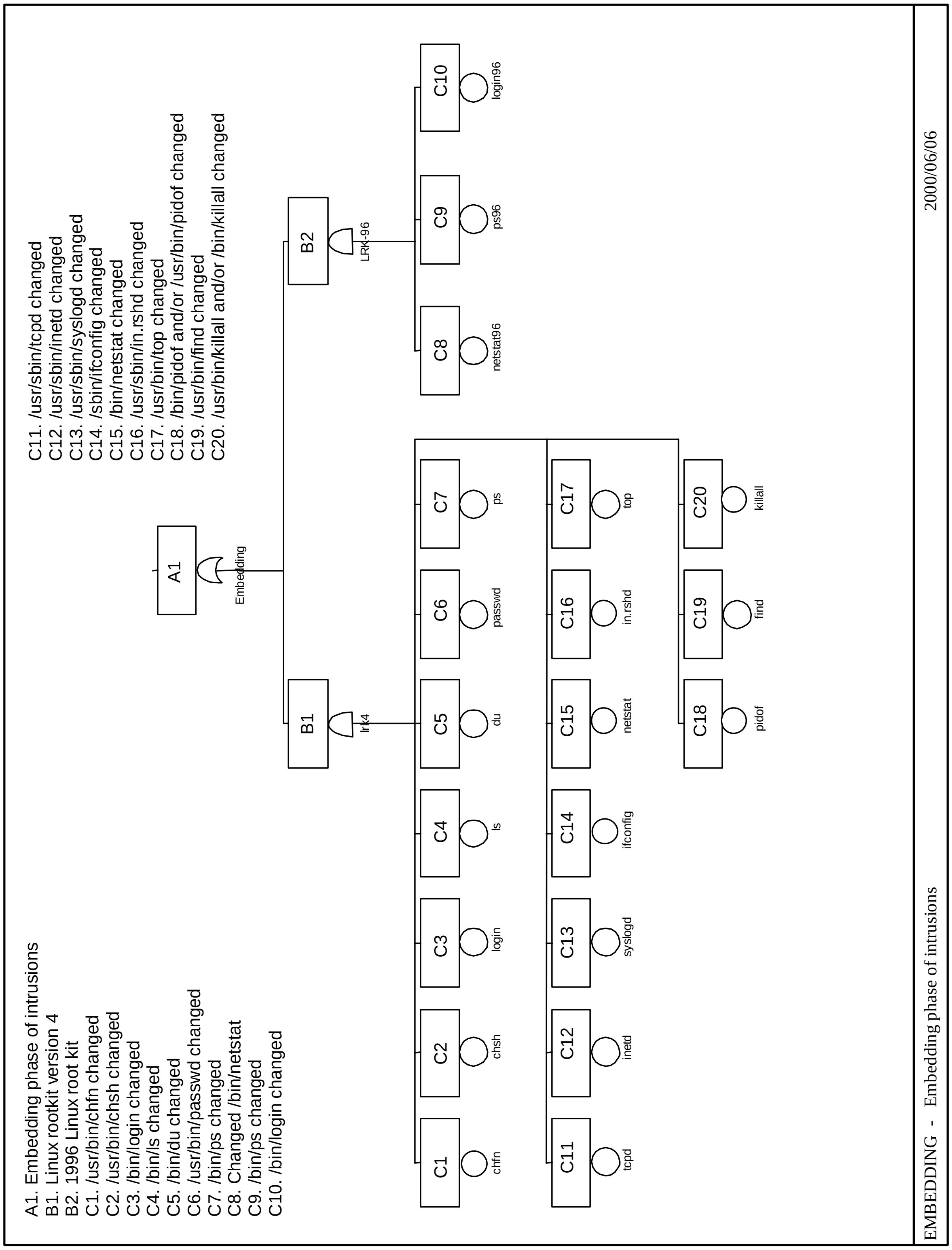

A1. Embedding phase of intrusions
B1. Linux rootkit version 4
B2. 1996 Linux root kit
C1. /usr/bin/chfn changed
C2. /usr/bin/chsh changed
C3. /bin/login changed
C4. /bin/ls changed
C5. /bin/du changed
C6. /usr/bin/passwd changed
C7. /bin/ps changed
C8. Changed /bin/netstat
C9. /bin/ps changed
C10. /bin/login changed
C11. /usr/sbin/tcpd changed
C12. /usr/sbin/inetd changed
C13. /usr/sbin/syslogd changed
C14. /sbin/ifconfig changed
C15. /bin/netstat changed
C16. /usr/sbin/in.rshd changed
C17. /usr/bin/top changed
C18. /bin/pidof and/or /usr/bin/pidof changed
C19. /usr/bin/find changed
C20. /usr/bin/killall and/or /bin/killall changed
A1
Embedding
B1
lrk4
B2
LRK-96
C1
chfn
C2
chsh
C3
login
C4
ls
C5
du
C6
passwd
C7
ps
C8
netstat96
C9
ps96
C10
login96
C11
tcpd
C12
inetd
C13
syslogd
C14
ifconfig
C15
netstat
C16
in.rshd
C17
top
C18
pidof
C19
find
C20
killall
EMBEDDING - Embedding phase of intrusions
2000/06/06

Figure 9 Embedding fault tree

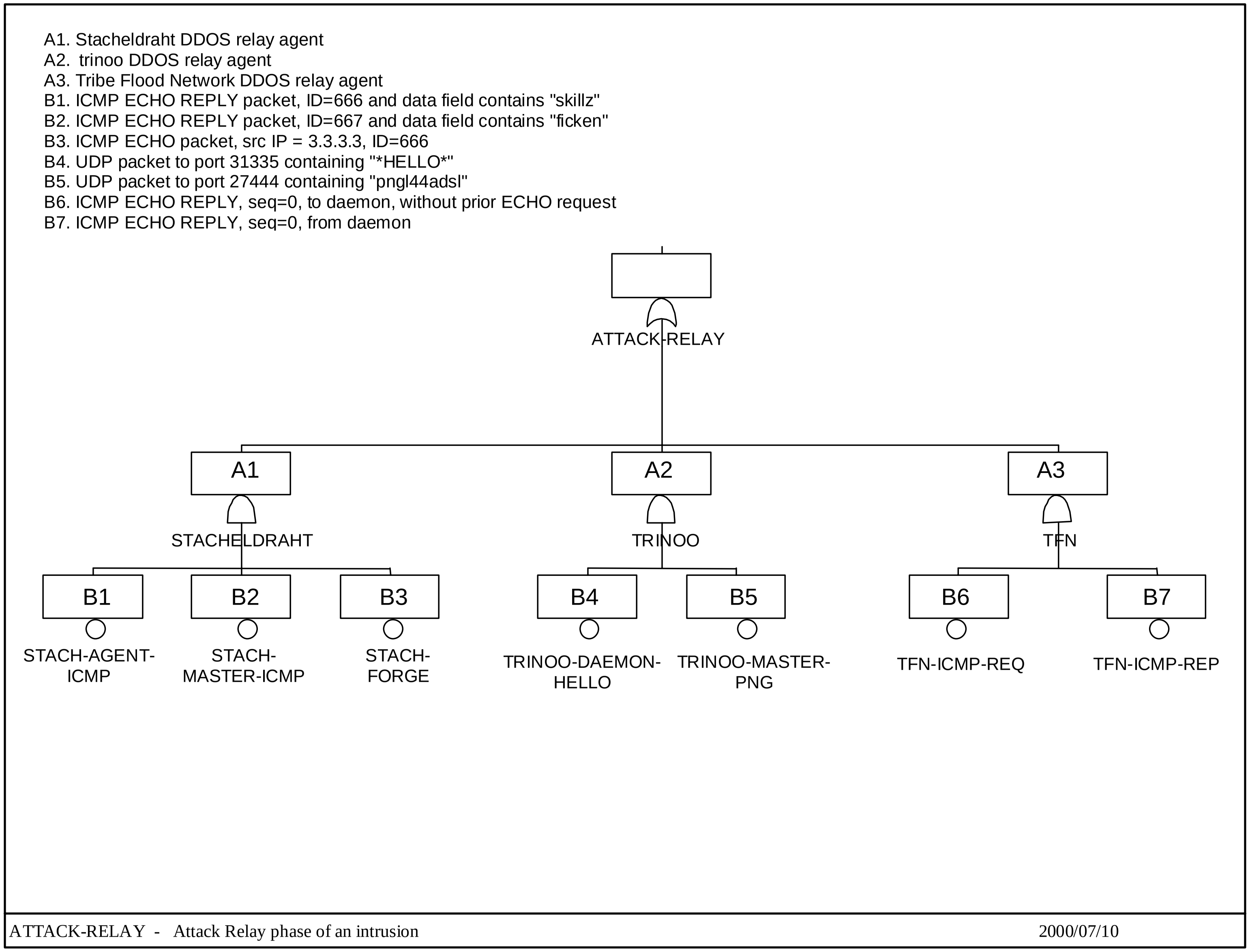

A1. Stacheldraht DDOS relay agent
A2. trinoo DDOS relay agent
A3. Tribe Flood Network DDOS relay agent
B1. ICMP ECHO REPLY packet, ID=666 and data field contains "skillz"
B2. ICMP ECHO REPLY packet, ID=667 and data field contains "ficken"
B3. ICMP ECHO packet, src IP = 3.3.3.3, ID=666
B4. UDP packet to port 31335 containing "*HELLO*"
B5. UDP packet to port 27444 containing "pngl44adsl"
B6. ICMP ECHO REPLY, seq=0, to daemon, without prior ECHO request
B7. ICMP ECHO REPLY, seq=0, from daemon
ATTACK-RELAY
A1
A2
A3
STACHELDRAHT
TRINOO
TFN
B1
B2
B3
B4
B5
B6
B7
STACH-AGENT-ICMP
STACH-MASTER-ICMP
STACH-FORGE
TRINOO-DAEMON-HELLO
TRINOO-MASTER-PNG
TFN-ICMP-REQ
TFN-ICMP-REP
ATTACK-RELAY - Attack Relay phase of an intrusion
2000/07/10

Figure 10 Attack relay fault tree